Introdução à Programação
uma Abordagem Funcional

Crediné Silva de Menezes,
Maria Claudia Silva Boeres,
Maria Christina Valle Rauber,
Thais Helena Castro,
Alberto Nogueira de Castro Júnior,
Cláudia Galarda Varassin

Departamento de Informática - UFES
Departamento de Ciência da Computação – UFAM
2008

Índice

1. CONCEITOS BÁSICOS

1.1 INTRODUÇÃO

Neste curso o leitor estará se envolvendo com a aprendizagem de conceitos e métodos básicos para a construção de programas de computador. A abordagem que daremos está voltada para o envolvimento do aprendiz com a solução de problemas ao invés da atitude passiva de ver o que os outros fizeram. Uma questão central que permeia o curso é a de que construir programas é uma tarefa de engenharia, e que, portanto produzirá artefatos com os quais o ser humano terá de conviver. Artefatos estes que devem satisfazer requisitos de qualidade e serem, portanto, passíveis de constatação.

Optamos desenvolver a disciplina orientada à descrição de funções, um formalismo bastante conhecido por todos os que chegam a este curso. Esperamos, com isso, atenuar algumas dificuldades típicas do ensino introdutório de programação. Nas seções seguintes apresentamos alguns conceitos básicos que nos parecem importantes ter em mente antes de iniciarmos um curso de programação.

1.2. COMPUTADORES

Denominamos computador uma máquina de processar dados, numéricos ou simbólicos, que funciona através da execução de programas. Ao contrário das inúmeras máquinas que conhecemos, tais como máquina de lavar roupa, liquidificador, aspirador de pó, e tantas outras, que realizam uma única função, o computador é uma máquina multi-uso. Podemos usá-lo como uma máquina de escrever sofisticada, como uma máquina de fax, como uma prancheta de desenho sofisticada, como um fichário eletrônico, como uma planilha de cálculos e de tantas outras formas. É exatamente como o nosso conhecido videogame: para mudar de jogo basta trocar o cartucho. No videogame, cada novo jogo é determinado por um novo programa.

Em linhas gerais podemos entender um computador como uma máquina capaz de:

a) interpretar dados que lhe são fornecidos, produzindo resultados em forma de novos dados, usando para isso conceitos que lhe foram antecipadamente informados e,
b) aceitar a descrição de novos conceitos e considerá-los na interpretação de novas situações.

Alguns exemplos de uso de um computador:

1) Descrever para uma máquina a relação métrica que existe entre os lados de um triângulo retângulo. De posse desse conhecimento, a máquina poderia, por exemplo, determinar o valor de um dos lados quando conhecido o valor dos outros dois.

2) Informar a uma máquina as regras de conjugação de verbos. Com este conhecimento a máquina pode determinar a forma correta para um determinado tempo e pessoa de um verbo específico.

3) Tradução de textos;

4) Classificação de textos quanto à sua natureza: romance, poesia, documentário, entrevista, artigo científico;

5) Manipulação de expressões algébricas, resolução de integral indefinida, etc;

6) Programação automática: dada uma certa especificação, gerar um programa eficiente;

7) Monitoramento de pacientes em um Centro de Tratamento Intensivo;

8) Identificação de tumores no cérebro a partir da comparação de imagens com padrões conhecidos de anormalidade;

9) Roteamento inteligente de mensagens;

10) Monitoramento de regiões por satélite.

1.3. PROGRAMAÇÃO

À tarefa de identificar o conhecimento necessário para a descrição de um conceito, organizá-lo e codificá-lo de modo a ser entendido pela máquina damos o nome de *programação de computadores*. Ao conhecimento codificado, produto final da tarefa de programação dá-se o nome de *programa*.

A programação de computadores é uma atividade que compreende várias outras atividades, tais como: entendimento do problema a ser resolvido, planejamento de uma solução, formalização da solução usando uma linguagem de programação, verificação da conformidade da solução obtida com o problema proposto.

1.4. LINGUAGEM DE PROGRAMAÇÃO

A descrição de conhecimento para um agente racional qualquer (seja uma máquina ou um humano) subentende a existência de padrões segundo os quais o agente possa interpretar o conhecimento informado. A esses padrões, quando rigorosamente elaborados, damos o nome de *formalismo*. Um formalismo é composto de dois aspectos: a sintaxe e a semântica. A sintaxe permite ao agente reconhecer quando uma "seqüência de símbolos" que lhe é fornecida está de acordo com as regras de escrita e, portanto representa um programa. A semântica permite que o agente atribua um significado ao conhecimento descrito pela "seqüência de símbolos". Por exemplo, quando um agente humano (com determinado grau de escolaridade) encontra a seguinte seqüência de símbolos {3, 4} U {5, 9, 15}, ele por certo reconhecerá como uma expressão algébrica escrita corretamente e, se lembrar dos fundamentos da

teoria dos conjuntos, associará esta cadeia como a descrição de um conjunto composto pela união dos elementos de dois conjuntos menores.

Eis aqui algumas observações importantes sobre a necessidade de linguagens de programação:

- Ainda não é possível usar linguagem natural para ensinar o computador a realizar uma determinada tarefa. A linguagem natural, tão simples para os humanos, possui ambigüidades e redundâncias que a inviabilizam como veículo de comunicação com os computadores.
- A linguagem nativa dos computadores é muito difícil de ser usada, pois requer do programador a preocupação com muitos detalhes específicos da máquina, tirando pois atenção do problema.
- Para facilitar a tarefa de programação foram inventadas as linguagens de programação. Estas linguagens têm por objetivo se colocarem mais próximas do linguajar dos problemas do que do computador em si. Para que o programa que escrevemos possa ser “entendido” pelo computador, existem programas especiais que os traduzem (compiladores) ou os que interpretam (interpretadores) para a linguagem do computador.
- Podemos fazer um paralelo com o que ocorre quando queremos nos comunicar com uma pessoa de língua estrangeira. Podemos escrever uma carta em nossa língua e pedir a alguém que a traduza para a língua de nosso destinatário ou se quisermos conversar pessoalmente, podemos usar um intérprete.

1.5. PROPRIEDADES DE UM PROGRAMA

Fazemos programas com a intenção de dotar uma máquina da capacidade de resolver problemas. Neste sentido, um programa é um produto bem definido, que para ser usado precisa que sejam garantidas algumas propriedades. Aqui fazemos referências a duas delas: a correção e o desempenho. A correção pode ser entendida como a propriedade que assegura que o programa descreve corretamente o conhecimento que tínhamos intenção de descrever. O desempenho trata da propriedade que assegura que o programa usará de forma apropriada o tempo e os recursos da máquina considerada. Cabe aqui alertar aos principiantes que a tarefa de garantir que um programa foi desenvolvido corretamente é tão complexa quanto à própria construção do programa em si. Garantir que um programa funciona corretamente é condição imprescindível para o seu uso e, portanto estaremos dando maior ênfase a esta propriedade.

1.6. PARADIGMAS DE LINGUAGENS DE PROGRAMAÇÃO

As regras que permitem a associação de significados às "seqüências de símbolos" obedecem a certos princípios. Existem várias manifestações destes princípios e a cada uma delas denominamos de paradigma.

Um paradigma pode ser entendido informalmente como uma forma específica de se "pensar" sobre programação. Existem três grandes grupos de

paradigmas para programação: o procedimental (ou procedural), o funcional e o lógico. Os dois últimos são freqüentemente referidos como sendo subparadigmas de um outro mais geral, o paradigma declarativo. O paradigma procedimental subentende a organização do conhecimento como uma seqüência de tarefas para uma máquina específica. O paradigma lógico requer o conhecimento de um formalismo matemático denominado lógica matemática. O paradigma funcional baseia-se no uso dos princípios das funções matemáticas. De uma forma geral, os paradigmas declarativos enfatizam o aspecto correção e o procedimental os aspectos de desempenho. Vejam que falamos em "enfatizam", o que quer dizer que apresentam facilidades para descrição e verificação da propriedade considerada. Entretanto, em qualquer caso, o programador deverá sempre garantir que os dois aspectos (correção e desempenho) sejam atendidos.

1.7. PROGRAMAÇÃO FUNCIONAL

Para os fins que aqui nos interessam neste primeiro momento, podemos entender o computador, de uma forma simplificada, como uma máquina capaz de:

a) avaliar expressões escritas segundo regras sintáticas bem definidas, como a das expressões aritméticas que tão bem conhecemos (ex. 3 + 5 - 8) obedecendo à semântica das funções primitivas das quais ela é dotada (por exemplo: as funções aritméticas básicas como somar, subtrair, multiplicar e dividir);

b) aceitar a definição de novas funções e posteriormente considerá-las na avaliação de expressões submetidas à sua avaliação.

Por enquanto, denominaremos o computador de máquina funcional. Na Figura 1.1 apresentamos um exemplo de interação de um usuário com a nossa Máquina Funcional.

usuário:	3 + 5 / 2
Máquina funcional::	5,5
usuário:	f x y = (x + y) / 2
Máquina funcional:	definição de **f** foi aceita
usuário:	(f 3 5) + (f 10 40)
Máquina funcional:	29

Figura 1.1

Na primeira interação podemos observar que o usuário descreveu uma expressão aritmética e que a máquina funcional avaliou e informou o resultado. Na segunda interação o usuário descreve, através de uma equação, uma nova função, que ele denominou de **f** e que a máquina funcional acatou a nova definição. Na terceira interação o usuário solicita a avaliação de uma nova expressão aritmética usando o conceito recentemente definido e que a máquina funcional faz a avaliação usando corretamente o novo conceito. Desta forma, percebemos que a máquina funcional é capaz de avaliar expressões

aritméticas e funções e também aceitar definições de funções, usando para isso, ambientes distintos.

1.8. EXPRESSÕES ARITMÉTICAS

A nossa máquina funcional hipotética entende a sintaxe das expressões aritméticas, com as quais todo aluno universitário já é bem familiarizado e é capaz de avaliá-las usando as mesmas que regras que já conhecemos.

Sintaxe - Todo operador aritmético pode ser entendido, e aqui o será, como uma função que possui dois parâmetros. A notação usual para as operações aritméticas é a infixada, ou seja, símbolo funcional colocado entre os dois operandos. Nada impede de pensarmos nelas escritas na forma prefixada, que é a notação usual para funções com número de parâmetros diferente de 2. Por exemplo, podemos escrever "***+ 3 2***" para descrever a soma do número 3 com o número 2. As funções definidas pelo programador devem ser escritas de forma prefixada, como no exemplo de interação apresentado na Figura 1.1. Combinando essas duas formas, infixada e prefixada, podemos escrever expressões bastante sofisticadas.

Avaliação - As expressões aritméticas, como sabemos, são avaliadas de acordo com regras de avaliação bem definidas, efetuando as operações de acordo com suas prioridades. Por exemplo, na expressão "***3 + 5 / 2***" o primeiro operador a ser avaliado será o de divisão (/) e posteriormente o de adição. Se desejarmos mudar essa ordem, podemos usar parênteses em qualquer quantidade, desde que balanceados e em posições apropriadas. Por exemplo, na expressão "***(3 + 5) / 2***", a utilização de parênteses determina que a sub-expressão 3 + 5 terá prioridade na avaliação.

1.9. FUNÇÕES

Podemos entender o conceito de funções como uma associação entre elementos de dois conjuntos A e B de tal forma que para cada elemento de A existe apenas um elemento de B associado. O conjunto A é conhecido como o domínio da função, ou ainda como o conjunto de entrada, e o conjunto B é o contra-domínio ou conjunto de saída. Para ser mais preciso, podemos afirmar que uma função *f* , que associa os elementos de um conjunto A aos elementos de um conjunto B, consiste em um conjunto de pares ordenados onde o primeiro elemento do par pertence a A o segundo a B. Exemplos:

a) Seja a função T que associa as vogais do alfabeto com os cinco primeiros inteiros positivos.

   T = {(a,1), (e,2), (i,3), (o,4), (u,5)}

b) Seja a função Q, que associa a cada número natural o seu quadrado.

   Q = {(0,0), (1,1), (2,4), (3,9), (4,16), ...}

Podemos observar que a função T é um conjunto finito e que a função Q é um conjunto infinito.

1.10. DESCRIÇÕES FUNCIONAIS

Podemos descrever um conjunto, de duas formas: *extensional*, onde explicitamos todos os elementos que são membros do conjunto, como no caso do conjunto T apresentado anteriormente; ou na forma *intencional*, onde descrevemos um critério de pertinência dos membros do conjunto. Por exemplo, o conjunto Q acima apresentado poderia ser reescrito da seguinte forma:

Q = {(x, y) | x é natural e y = x.x} que pode ser lido da seguinte maneira:

Q é o conjunto dos pares ordenados (x, y) tal que x é um número natural e y é o produto de x por x.

Quando descrevemos uma função para fins computacionais, estamos interessados em explicitar como determinar o segundo elemento do par ordenado, conhecido o primeiro elemento do par. Em outras palavras, como determinar y conhecendo-se o valor de x. Normalmente dizemos que queremos determinar y em função de x. Nesta forma de descrição, omitimos a variável y e explicitamos o primeiro elemento que é denominado então de parâmetro da função. No caso acima teríamos então:

Q *x* = *x* . *x*

1.11. POR QUE COMEÇAR A APRENDIZAGEM DE PROGRAMAÇÃO ATRAVÉS DO PARADIGMA FUNCIONAL?

Tendo em vista a prática vigente de começar o ensino de programação em cursos de computação utilizando o paradigma procedimental, apresentamos a seguir alguns elementos que baseiam nossa opção de começar o ensino de programação usando o paradigma funcional.

1) O aluno de graduação em Computação tem de 4 a 5 anos para aprender todos os detalhes da área de computação, portanto não se justifica que tudo tenha que ser absorvido no primeiro semestre. O curso introdutório é apenas o primeiro passo e não visa formar completamente um programador. Este é o momento de apresentar-lhe os fundamentos e, além disso, permitir que ele vislumbre a variedade de problemas que podem ser solucionados como o apoio do computador;

2) O paradigma procedimental requer que o aluno tenha um bom entendimento dos princípios de funcionamento de um computador real, pois eles se baseiam, como as máquinas reais, no conceito de mudança de estados (máquina de Von Neumann).

3) O paradigma lógico, outro forte candidato, requer o conhecimento de lógica matemática que o aluno ainda não domina adequadamente;

4) O paradigma funcional é baseado num conhecimento que o aluno já está familiarizado desde o ensino médio (funções, mapeamento entre domínios) o qual é ainda explorado em outras disciplinas do

ciclo básico, o que nos permite concentrar nossa atenção na elaboração de soluções e na descrição formal destas;

5) O elevado poder de expressão das linguagens funcionais permite que a abrangência do uso do computador seja percebida mais rapidamente. Em outras palavras, podemos resolver problemas mais complexos já no primeiro curso;

6) O poder computacional do paradigma funcional é idêntico ao dos outros paradigmas. Apesar disso, ele ainda não é usado nas empresas, por vários aspectos. Dentre os quais podemos citar:

   i) No passado, programas escritos em linguagens funcionais eram executados muito lentamente. Apesar disso não ser mais verdadeiro, ficou a fama;

   ii) A cultura de linguagens procedimentais possui muitos adeptos no mundo inteiro, o que, inevitavelmente, cria uma barreira à introdução de um novo paradigma. Afinal, temos medo do desconhecido e trememos quando temos que nos livrar de algo que já sabemos;

   iii) Há uma crença que linguagens funcionais são difíceis de aprender e só servem para construir programas de inteligência artificial.

7) A ineficiência das linguagens funcionais em comparação às procedimentais tem se reduzido através de alguns mecanismos tais como: lazy evaluation, grafo de redução, combinadores.

8) Para fazer um programa que "funciona" (faz alguma coisa, embora não necessariamente o que desejamos) é mais fácil fazê-lo no paradigma procedimental. Para fazer um programa que funciona "corretamente" para resolver um determinado problema é mais fácil no paradigma funcional, pois esse paradigma descreve "o que fazer" e não "como fazer".

9) As linguagens funcionais são geralmente utilizadas para processamento simbólico, ou seja, solução de problemas não numéricos. (Exemplo: integração simbólica X integração numérica). Atualmente constata-se um crescimento acentuado do uso deste tipo de processamento.

10) A crescente difusão do uso do computador nas mais diversas áreas do conhecimento gera um crescimento na demanda de produção de programas cada vez mais complexos. Os defensores da programação funcional acreditam que o uso deste paradigma seja uma boa resposta a este problema, visto que com linguagens funcionais podemos nos concentrar mais na solução do problema do que nos detalhes de um computador específico, o que aumenta a produtividade.

11) Se mais nada justificar o aprendizado de uma linguagem funcional como primeira linguagem, resta a explicação didática. Estamos dando um primeiro passo no entendimento de programação como um todo. É, portanto, importante que este

passo seja simplificado através do apoio em uma "máquina" já conhecida, como é o caso das funções.

12) Ainda um outro argumento: mesmo que tenhamos que usar posteriormente uma outra linguagem para ter uma implementação eficiente, podemos usar o paradigma funcional para formalizar a solução. Sendo assim, a versão em linguagem funcional poderia servir como uma especificação do programa.

Exercícios

1. Conceitue programação de computadores.
2. Quais os principais paradigmas de programação e o que os diferenciam.
3. Faça uma descrição intencional da função: F = {1,3,5,7,...}.
4, Faça uma listagem de outros exemplos de programas de computador que são usados hoje em diferentes áreas do conhecimento e por diferentes profissionais.
5. Apresente exemplo de outras linguagens técnicas usadas pelo ser humano para descrever conhecimento.
6. Os conceitos de correção e de desempenho, se aplicam a qualquer artefato. Escolha 3 artefatos quaisquer e discuta os dois conceitos.
7. Apresente uma descrição informal da função que conjuga os verbos regulares da 1ª. conjugação.

2. A LINGUAGEM DE PROGRAMAÇÃO HASKELL E O AMBIENTE HUGS

2.1. INTRODUÇÃO

Neste curso usaremos o ambiente HUGS, no qual é utilizada uma implementação da linguagem de programação funcional Haskell. Essa linguagem por apresentar uma sintaxe simples e elegante, além de oferecer operadores bastante expressivos, tem sido usada com bons resultados para a aprendizagem de fundamentos de programação.

Podemos usar o HUGS como uma calculadora qualquer, à qual submetemos expressões que ela avalia e nos informa o valor resultante. Vejamos por exemplo as interações a seguir.

```
? 3 + 5 * 2
13
? (3 + 5) * 2
16
?
```

Nas expressões acima é importante destacar o uso do símbolo * (asterisco) que é empregado para representar a multiplicação. Além desse, outros símbolos usuais serão substituídos, como veremos logo mais. As operações aritméticas são, como sabemos, funções. A notação utilizada em Haskell para as operações aritméticas é a usual, ou seja, infixada (o símbolo da operação fica entre os operandos). Uma outra forma existente em Haskell para escrever expressões é usando o operador de forma prefixada. Por exemplo,a expressão ***div 15 6 ,*** indica a divisão inteira do número 15 (dividendo) pelo número 6 (divisor).

No ambiente HUGS, o símbolo **>** é usado para indicar que o sistema está preparado para avaliar uma nova expressão. Após avaliar uma expressão o resultado é informado na linha seguinte e em seguida uma nova interrogação é exibida, se disponibilizando para uma nova avaliação. O avaliador de expressões do ambiente HUGS funciona conforme o esquema da figura 2.1. Repetidamente o ambiente HUGS permite a leitura de uma expressão, sua avaliação e exibição do resultado.

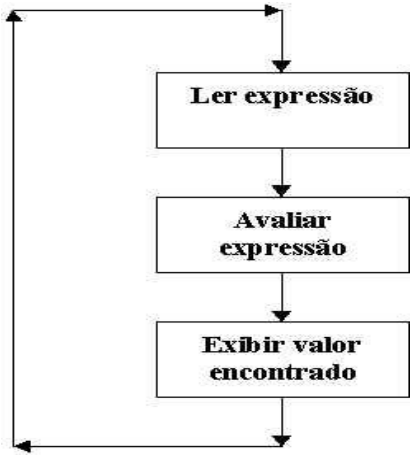


Figura 2.1

Podemos usar o ambiente HUGS somente para isso, para escrever expressões e solicitar ao sistema que as avalie. Entretanto podemos ousar mais e usar o ambiente para descrever novas funções a partir das funções já oferecidas pelo ambiente. As funções já

oferecidas pelo ambiente são conhecidas como **primitivas**. Podemos também utilizar nessas novas definições de funções aquelas que tivermos construído anteriomente.

2.2. DESCRIÇÃO DE FUNÇÕES

A forma de descrever funções é similar ao que nos referimos anteriormente no Capítulo 1, ou seja, através de uma equação, onde no lado esquerdo da igualdade definimos um nome para a função e relacionamos os parâmetros (ou argumentos) considerados na sua definição. No lado direito escrevemos uma expressão utilizando outras funções, primitivas ou não. Isto nos leva portanto a tomar conhecimento que a linguagem possui funções primitivas que já a acompanham e que portanto prescindem de definição.

Por exemplo, para definir a função que determina o espaço percorrido por um móvel em movimento retilíneo uniforme, conhecidos a sua velocidade e o tempo decorrido, podemos escrever:

espaco v t = v * t

No esquema a seguir fazemos uma identificação didática dos vários elementos da definição. O lado esquerdo da igualdade também é chamado de inteface ou assinatura da função e o lado direito o corpo da definição.

espaco	v t	=	v * t
nome da função	parâmetros		expressão aritmética que define a relação que há entre os parâmetros
interface da função			corpo da definição

Para alimentar o HUGS com novas definições devemos criar um arquivo texto em disco no qual editaremos as definições desejadas. Cada arquivo pode ter uma ou mais definições de funções. Normalmente agrupamos em um mesmo arquivo as definições relacionadas com a solução de um problema específico ou definições de propósito geral. No jargão de programação com a linguagem Haskell, um conjunto de definições é denominado **script**.

A alimentação de novas definições é indicada ao sistema através de um comando que é usado no lugar de uma expressão, quando o sistema exibe o seu pedido de tarefa (o símbolo de interrogação). Para indicar que estaremos executando um comando, usamos o símbolo " **:** " (dois pontos) seguido do nome do comando (existem vários). Para a tarefa que temos neste instante utilizamos o comando ***load*** (oportunamente veremos outros).

Por exemplo, podemos escrever um conjunto de definições em um arquivo denominado **pf001.hs**. Para alimentar este arquivo no ambiente HUGS (interpretador) podemos escrever:

> :load pf001.hs

A partir deste momento, todas as definições contidas no arquivo informado estarão disponíveis para uso. Isso pode ser entendido como uma extensão da máquina Haskell. Na Figura 2.2 apresentamos um esquema de utilização do ambiente HUGS e de um editor de textos para provimento de novas definições.

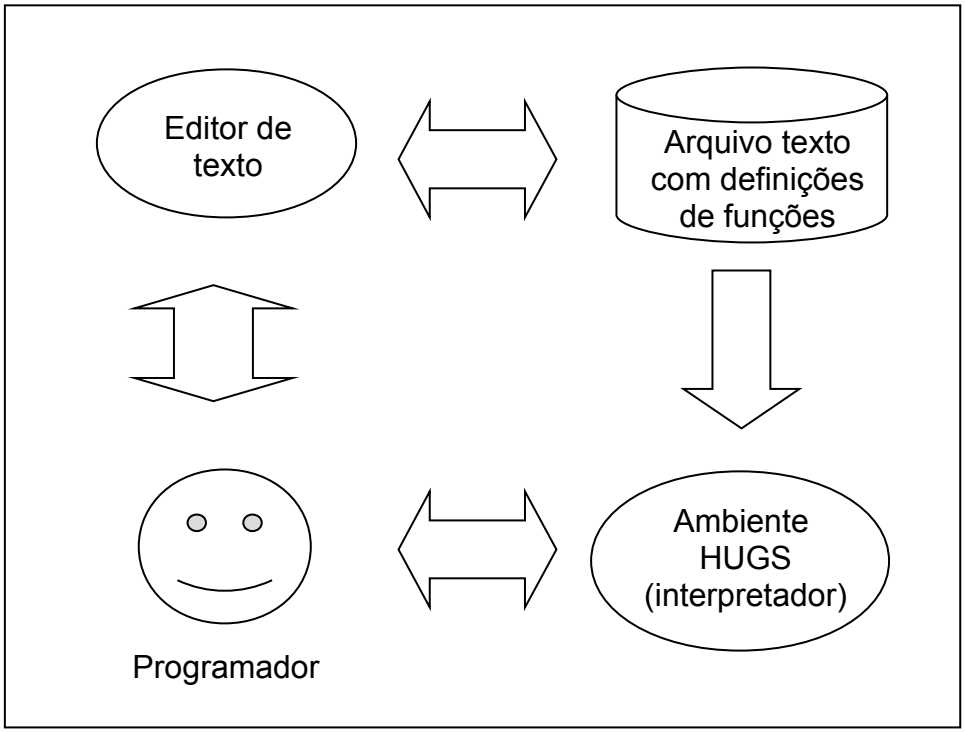


Figura 2.2 – Interações do Programador com o Ambiente de Programação

Vale aqui um pequeno lembrete, podemos estender a noção de funções para incorporar também as chamadas funções constantes. Na nomenclatura de definições de funções, dizemos que temos definições paramétricas e definições não paramétricas.

Por exemplo, podemos definir a constante ***pi***, da seguinte maneira:

```
pi = 3.1416
```

A constante pi pode ser entendida como uma função não paramétrica.

2.3. UM EXEMPLO

Considere que queremos descrever uma função para determinar as raízes de uma equação do segundo grau.Sabemos que pela nossa clássica fórmula as raízes são descritas genericamente por:

$$x = \frac{-b \pm \sqrt{b^2 - 4 \times a \times c}}{2 \times a}$$

A solução, como sabemos, é formada por um par de valores. Por enquanto vamos descrever este fato por duas funções, uma para a primeira raiz e outra para a segunda.

```
eq2g1 a b c = ((-b) + sqrt (b^2 - 4.0 * a * c )) / (2.0 * a)
```

Vamos discutir alguns detalhes desta codificação:

- o termo ***-b*** precisa ser codificado entre parêntesis pois números negativos são obtidos por um operação unária que produz um número negativo a partir de um positivo;
- o símbolo da raiz quadrada foi substituído pela função ***sqrt***;
- o numerador da fração precisa ser delimitado pelo uso de parêntesis;
- o denominador também precisa ser delimitado por parêntesis;
- o símbolo de multiplicação usual foi trocado pelo * (asterisco);
- o símbolo de potenciação foi substituído pelo ^ (circunflexo).

Podemos agora descrever a outra raiz de forma análoga:

*eq2g2 a b c = ((-b) - sqrt (b^2 - 4.0 * a * c)) / (2.0 * a)*

Visto que as duas descrições possuem partes comuns, poderíamos ter escrito abstrações auxiliares e produzir um conjunto de definições, tal como:

quad x	=	*x * x*
raizdelta a b c	=	*sqrt (quad b - 4.0 * a * c)*
dobro x	=	*2.0 * x*
eq2g1 a b c	=	*((-b) + raizdelta a b c) / dobro a*
eq2g2 a b c	=	*((-b) - raizdelta a b c) / dobro a*

Vejamos como ficaria uma interação com o HUGS, a partir de um arquivo de definições denominado **eq2g.hs**:

*> **:l** eq2g.hs*
Reading script file "eq2g.hs":

Haskell session for:
standard.prelude
eq2g.hs

> eq2g1 2.0 5.0 2.0
-0.5
> eq2g2 2.0 5.0 2.0
-2.0
> eq2g1 3.0 4.0 5.0

Program error: {sqrt (-44.0)}

Podemos observar que houve um problema com a avaliação da expressão

eq2g1 3.0 4.0 5.0

Este é um erro semântico, provocado pela tentativa de extrair a raiz quadrada de um número negativo. A função que definimos portanto é uma função parcial, ou

seja, ela não está definida para todo o domínio dos reais. Neste caso o sistema apenas acusa o problema ocorrido.

2.4. DEFINIÇÕES LOCAIS

As definições que discutimos anteriormente são globais, no sentido de que estão acessíveis ao uso direto do usuário e também disponíveis para uso em qualquer outra definição. Por exemplo, se temos o ***script***

```
quad x  =  x * x
hipo x y = sqrt ( quad x + quad y)
```

podemos utilizar a definição ***quad*** tanto externamente pelo usuário quanto internamente pela definição ***hipo***.

Se no entanto desejamos construir subdefinições para uso em uma definição específica, podemos definí-las internamente, restringindo o seu contexto. A maneira de introduzir definições locais em Haskell é por meio da cláusula ***where.*** Vamos modificar o script acima para que ***quad*** só possa ser usado no interior de ***hipo***.

hipo x y	=	sqrt (quad x + quad y)
		where
		quad x = x * x

As definições internas também não precisam ser paramétricas. Veja este outro exemplo.

hipo x y	=	sqrt (k1 + k2)
		where
		k1 = x * x k2 = y * y

Note que apesar de ***x*** e ***y*** não serem parâmetros de ***k1*** e ***k2*** eles foram utilizados em suas definições. Isto é possível porque ***x*** e ***y*** têm validade em todo o lado direito da definição.

Temos que considerar ainda que nada impede que em uma definição local tenhamos uma outra definição local e dentro desta outra, e assim sucessivamente.

hipo x y	=	sqrt k	
		where	
		k = quad x + quad y	
			where
			quad x = x * x

2.5. MODELO DE AVALIAÇÃO DE EXPRESSÕES: Quando o avaliador (interpretador) do HUGS toma uma expressão contendo apenas constantes e operações primitivas, ele apenas efetua as operações obedecendo à prioridade dos operadores e aquelas determinadas pelo uso de parêntesis. Por exemplo, para avaliar a expressão 3 + 5 / 2,

primeiro é realizada a divisão 5/2, resultando em 2.5, o qual é adicionado ao 3, finalmente obtendo 5.5. Podemos entender isso como uma seqüência de reduções de uma expressão à outra mais simples, até que se atinja um termo irredutível.
Veja os exemplos a seguir:

3 + 5 / 2 ➔ 3 + 2.5 ➔ 5.5
(3 + 5) / 2 ➔ 8 / 2 ➔ 4
3 + 5 + 2 ➔ 8 + 2 ➔ 10

As setas (➔) são usadas para indicar um passo de redução.

Quando as expressões utilizam funções definidas pelo usuário, o processo é análogo. Cada referência a uma definição é substituída por seu lado direito, até que se atinja uma expressão básica, e prossegue como no caso anterior. Vejamos os exemplos abaixo, considerando a primeira definição de ***hipo*** e de ***quad*** apresentada anteriormente:

ordem	expressão	redução aplicada
1	hipo 3 5 + hipo 4 4	expressão inicial
2	sqrt (quad 3 + quad 5) + hipo 4 4	def de hipo
3	sqrt (3 * 3 + quad 5) + hipo 4 4	def de quad
4	sqrt (3 * 3 + 5 * 5) + hipo 4 4	def de quad
5	sqrt (3 * 3 + 5 * 5) + sqrt (quad 4 + quad 4)	def de hipo
6	sqrt (3 * 3 + 5 * 5) + sqrt (4 * 4 + quad 4)	def de quad
7	sqrt (3 * 3 + 5 * 5) + sqrt (4 * 4 + 4 * 4)	def de quad
8	sqrt (9 + 5 * 5) + sqrt (4 * 4 + 4 * 4)	*
9	sqrt (9 + 25) + sqrt (4 * 4 + 4 * 4)	*
10	sqrt 34 + sqrt (4 * 4 + 4 * 4)	+
11	5.83095 + sqrt (4 * 4 + 4 * 4)	sqrt
12	5.83095 + sqrt (16 + 4 * 4)	*
13	5.83095 + sqrt (16 + 16)	*
14	5.83095 + sqrt (32)	+
15	5.83095 + 5.65685	sqrt
16	11.4878	+

Para uma realização específica da linguagem, isso não precisa acontecer exatamente assim. Entretanto este modelo é suficiente para nossos interesses. O número de reduções necessárias para chegar à forma irredutível de uma expressão, também denominada de forma canônica ou ainda forma normal, pode ser usado como critério para discutir o desempenho da mesma.

Exercícios

1. Estabeleça categorias para comparar duas definições;
2. Usando as categorias definidas no item 1 compare as duas definições apresentadas para as raízes de uma equação do segundo grau;
3. Compare usando as suas categorias, as três definições apresentadas para hipo;
4. Apresente uma explicação para o erro produzido pela avaliação da expressão ***eq2g1 3.0 4.0 5.0***
5. Apresente uma seqüência de reduções para a expressão: ***eq2g1 2.0 5.0 2.0***
6. Apresente um conjunto de definições para a função total para a determinação das raízes de uma equação do segundo grau.

2.6. ASSINATURA DE FUNÇÕES E A NOTAÇÃO CURRY:

Do estudo de funções sabemos que toda função possui um domínio e um contradomínio. O domínio pode ser formado por zero ou mais conjuntos. Quando uma função não possui domínio dizemos que é uma função constante. Para descrever este conhecimento sobre domínio e contradomínio é usado o conceito de "assinatura", do inglês "*signature*".

Para conhecer o tipo de uma função, disponível na biblioteca do HUGS ou construída pelo programador, basta usar, no ambiente HUGS, o comando:

Hugs> :t <nome da função>

O quadro a seguir apresenta alguns exemplos.

```
Hugs> :t sqrt
sqrt :: Floating a => a -> a
Hugs> :t sin
sin :: Floating a => a -> a
Hugs> :t abs
abs :: Num a => a -> a
Hugs> :t mod
mod :: Integral a => a -> a -> a
Hugs> :t div
div :: Integral a => a -> a -> a
```

O exemplo **`sqrt :: Floating a => a -> a`**, pode ser lido da seguinte maneira:

"A função **sqrt** mapeia um valor **a** do tipo **Floating** em outro valor do tipo **Floating**."

Os termos ***Integral***, ***Int***, ***Num***, ***Floating*** etc serão cuidadosamente apresentados no Capitulo 5. Por enquanto, basta antecipar que cada valor, inclusive os numéricos, podem ser de tipos diferentes e a linguagem provê uma classificação para esses valores. Podemos fazer uma analogia com a matemática clássica onde os números podem ser: ***naturais***, ***inteiros***, ***racionais***, ***reais***, ***irracionais***, ***complexos*** etc. As linguagem de programação, por questões relacionadas com a tecnologia digital, podem "inventar" outros tipos para facilitar a manipulação de dados pelos computadores digitais.

Vejamos um exemplo: na função que determina a média aritmética de 3 números reais, temos que o domínio é formado pelo produto cartesiano **R** x **R** X **R** e o contradomínio é **R**.

A ***assinatura*** desta função é a seguinte:

ma3 :: R X R X R → R

Em Haskell poderíamos ter a seguinte definição:

ma3 x y z = (x + y + z) / 3.

É usual também dizer que **R X R X R → R** é o tipo da função ***ma3***.

Para simplificar o trabalho com funções o matemático Haskell Curry criou uma nova notação, que considera que qualquer função tem sempre um único parâmetro de entrada. Segundo ele, o resultado de aplicar uma função sobre um parâmetro produz uma nova função que pode ser aplicada a outro parâmetro, e assim sucessivamente. Por exemplo, nesta notação, a função ***ma3*** teria a seguinte assinatura:

ma3 :: R → R → R → R

Podemos ler da seguinte maneira: ***ma3*** é uma função que mapeia um numero real em uma nova função cuja assinatura é:

ma3' :: R → R → R.

A função **ma3'** por sua vez mapeia um número real em uma nova função cuja assinatura é:

ma3'' :: R → R.

A função **ma3''** por sua vez mapeia um número real em uma nova função cuja assinatura é:

ma3''' :: R.

Que é uma função constante.

No exemplo acima, quando solicitamos a avaliação da expressão: **ma3 3 4 5**

o processo de avaliação pode ser entendido da seguinte maneira:

ma3 3 4 5 → "(x + y + z)/ 3" 4 5

```
→ “( 3 + y + z)/ 3” 4 5
→ “(3 + 4 + z)/3” 5
→ “(3 + 4 + 5)/3”
→ “(7 + 5)/3”
→ “12/3”
→ 4
```

Podemos ainda, obter a definição de novas funções a partir de uma função **f** previamente definida.

f x y z	=	x + y + z
g x y	=	f 3 x y
h x	=	g 4 x
m	=	h 5

No quadro a seguir mostramos a avaliação de tipos de cada uma das funções.

```
Main> :t f
f :: Num a => a -> a -> a -> a
Main> :t g
g :: Num a => a -> a -> a
Main> :t h
h :: Num a => a -> a
Main> :t m
m :: Integer
```

Vejamos agora uma avaliação de expressões com estas funções:

```
Main> (f 10 20 30) + (g 20 30) + (h 30) + m
162
Main> g 20 30
53
Main> f 10 20 30
60
Main> g 20 30
53
Main> h 30
37
Main> m
12
```

Exercícios

1) Avalie as expressões abaixo e apresente a sequência de reduções necessárias para a obtenção do termo irredutível (resultado final):
   a. mod 15 2
   b. mod 15 2 + div 6 3
   c. ma3 5 10 2

d. sqrt (15 – 2*3) / (17 – 12)

2) Escreva um arquivo texto (script) contendo a definição das funções abaixo. Use, quando for adequado, definições locais:
   a. Determinação da área de um retângulo de lados a e b
   b. Determinação da área de um círculo de raio r
   c. Determinação da média aritmética de três números a, b e c

3. A ARTE DE RESOLVER PROBLEMAS

3.1. INTRODUÇÃO

O grande objetivo deste curso é criar oportunidades para que o estudante desenvolva suas habilidades como resolvedor de problemas. Mais especificamente estamos interessados na resolução de problemas usando o computador, entretanto, temos certeza, que as idéias são gerais o suficiente para ajudar a resolver qualquer problema.

Embora muitas pessoas já tenham discutido sobre este assunto ao longo da história, nosso principal referencial será o professor George Polya, que escreveu um livro sobre este assunto, com o mesmo nome deste capítulo, voltado para a resolução de problemas de matemática do ensino fundamental. Todos os alunos deste curso terão grande proveito se lerem este livro.

As idéias ali apresentadas com certeza se aplicam com muita propriedade à resolução de problemas com o computador. Na medida do possível estaremos apresentando aqui algumas adaptações que nos parecem apropriadas neste primeiro contato com a resolução de problemas usando o computador.

3.2. DICAS INICIAIS

Apresenta-se a seguir algumas orientações:

3.2.1. Só se aprende a resolver problemas através da experiência. Logo o aluno que está interessado em aprender deve trabalhar, buscando resolver por si mesmo, os problemas propostos antes de tentar o auxílio do professor ou de outro colega;

3.2.2. A ajuda do professor não deve vir através da apresentação pura e simples de uma solução. A ajuda deve vir através do fornecimento de pistas que ajudem o aluno a descobrir a solução por si mesmo;

3.2.3. É muito importante não se conformar com uma única solução. Quanto mais soluções encontrar, mais hábil o estudante ficará, e além disso, poderá comparar as várias alternativas e escolher a que lhe parecer mais apropriada. A escolha deverá sempre ser baseada em critérios objetivos.

3.2.4. Na busca pela solução de um problema, nossa ferramenta principal é o questionamento. Devemos buscar formular questões que nos ajudem a entender o problema e a elaborar a solução.

3.2.5. Aprenda desde cedo a buscar um aprimoramento da sua técnica para resolver problemas, crie uma sistematização, evite chegar na frente de um problema como se nunca tivesse resolvido qualquer outro problema. Construa um processo individual e vá aperfeiçoando-o à cada vez que resolver um novo problema.

3.3 COMO RESOLVER UM PROBLEMA

O professor Polya descreve a resolução de um problema como um processo complexo que se divide em quatro etapas, as quais apresentamos aqui, com alguma adaptação. Segundo nos recomenda Polya, em qualquer destas etapas devemos ter em mente três questões sobre o andamento do nosso trabalho, que são: Por onde começar esta etapa? O que posso fazer com os elementos que disponho? Qual a vantagem de proceder da forma escolhida?

Etapa 1 - Compreensão do Problema

É impossível resolver um problema sobre o qual não tenhamos um entendimento adequado. Portanto, antes de correr atrás de uma solução, concentre-se um pouco em identificar os elementos do problema. Faça perguntas tais como: Quais são os dados de entrada? O que desejamos produzir como resultado? Qual a relação que existe entre os dados de entrada e o resultado a ser produzido? Quais são as propriedades importantes que os dados de entrada possuem?

Etapa 2 – Planejamento

Nesta fase devemos nos envolver com a busca de uma solução para o problema proposto. Pode ser que já o tenhamos resolvido antes, para dados ligeiramente modificados. Se não é o caso, pode ser que já tenhamos resolvido um problema parecido. Qual a relação que existe entre este problema que temos e um outro problema já conhecido? Será que podemos quebrar o problema em problemas menores? Será que generalizando o problema não chegaremos a um outro já conhecido? Conhecemos um problema parecido, embora mais simples, o qual quando generalizado se aproxima do que temos?

Etapa 3 - Desenvolvimento

Escolhida uma estratégia para atacar o problema, devemos então caminhar para a construção da solução. Nesta fase, devemos considerar os elementos da linguagem de programação que iremos usar, respeitando os elementos disponibilizados pela linguagem, tais como tipos, formas de definição, possibilidades de generalização e uso de elementos anteriormente definidos. A seguir, devemos codificar cada pedaço da solução, e garantir que cada um esteja descrito de forma apropriada. Em outras palavras, não basta construir, temos que ser capazes de verificar se o resultado de nossa construção está correto. Isto pode ser realizado pela identificação de instâncias típicas, da determinação dos valores esperados por estas, da submissão dessas instâncias ao avaliador e finalmente, da comparação dos resultados esperados com os resultados produzidos pela avaliação de nossas definições. Além disso, devemos garantir que a definição principal também está correta.

Em síntese, a fase de desenvolvimento compreende as seguintes subfases:

1. construção da solução;
2. planejamento do teste;
3. execução manual do teste;

4. codificação da solução;

5. teste com o uso do computador.

Etapa 4 - Avaliação do Processo e seus resultados

O trabalho do resolvedor de problemas não pára quando uma solução está pronta. Devemos avaliar a qualidade da solução, o processo que realizamos e questionar as possibilidades de uso posterior da solução obtida e também do próprio método utilizado. Novamente devemos fazer perguntas: Este foi o melhor caminho que poderia ter sido usado? Será que desdobrando a solução não obtenho componentes que poderei usar mais facilmente no futuro? Se esta solução for generalizada é possível reusá-la mais facilmente em outras situações? Registre tudo, organize-se para a resolução de outros problemas. Anote suas decisões, enriqueça a sua biblioteca de soluções e métodos.

3.4. UM PEQUENO EXEMPLO

Enunciado: Deseja-se escrever um programa que permita determinar a menor quantidade de cédulas necessárias para pagar uma dada quantia em Reais.

Etapa 1 – Compreensão

Questão: Quais os dados de entrada?
Resposta: A quantia a ser paga.

Questão: Qual o resultado a ser obtido?
Resposta: A menor quantidade de cédulas.

Questão: Qual a relação que existe entre a entrada e a saída?
Resposta: O somatório dos valores de cada cédula utilizada deve ser igual à quantia a ser paga.

Questão: Existe algum elemento interno, ou seja, uma dado implícito?Resposta: Sim, os tipos de cédulas existentes.Considere que o Real possui apenas as seguintes cédulas: 1, 5, 10, 50 e 100. É importante observar que qualquer cédula é um múltiplo de qualquer uma das menores.

Etapa 2 – Planejamento

Conheço algum problema parecido? Existe um problema mais simples?

Podemos entender como um problema mais simples um que busque determinar quantas cédulas de um dado valor são necessárias para pagar a quantia desejada. Por exemplo, para pagar R$ 289,00 poderíamos usar 289 cédulas de R$ 1,00. Ou também poderíamos usar 5 notas de R$ 50,00, mas neste caso ficariam ainda faltando R$ 39,00. Claro que não queremos que falte nem que sobre e, além disso,

desejamos que a quantidade seja mínima. Parece uma boa estratégia é começar vendo o que dá para pagar com a maior cédula e determinar quando falta pagar. O restante pode ser pago com uma cédula menor, e por aí afora. Explorando a instância do problema já mencionada, vamos fazer uma tabela com estes elementos.

Expressão para determinar a quantidade de cédulas de um determinado valor.	Quantidade de cédulas	Quantia a Pagar
		289,00
289 / 100	2	89,00
89/ 50	1	39,00
39/ 10	3	9,00
9/ 5	1	4,00
4/ 1	4	0,00
TOTAL	**11**	

Etapa 3 - Desenvolvimento

Solução 1 - Considerando a tabela acima descrita, codificando cada linha como uma subexpressão da definição.

```
ncedulas q  =  (div q 100) +
               (div (mod q 100) 50) +
               (div (mod (mod q 100) 50) 10) +
               (div (mod (mod (mod q 100) 50) 10) 5)+
               (div (mod (mod (mod (mod q 100) 50) 10) 5) 1)
```

Solução 2 - Considerando uma propriedade das cédulas, ou seja, já que uma cédula qualquer é múltiplo das menores, a determinação do resto não precisa considerar as cédulas maiores do que a cédula que estamos considerando em um dado ponto.

```
ncedulas2 q  =  (div q 100) +
                (div (mod q 100) 50) +
                (div (mod q  50) 10) +
                (div (mod q  10) 5)+
                (div (mod q 5)  1)
```

Etapa 4 – Avaliação

A solução deixa de explicitar as abstrações referentes à quantidade de cédulas de um determinado valor, assim como o resto correspondente. Podemos questionar, não seria melhor explicitar? Assim poderíamos usá-las de forma independente e além disso, a solução fica mais clara e portanto inteligível.

```
ncedulas q = n100 q + n50 q + n10 q + n5 q + n1 q
n100 q     = div q 100
r100 q     = mod q 100
n50 q      = div (r100 q) 50
r50 q      = mod (r100 q) 50
n10 q      = div (r50 q) 10
r10 q      = mod (r50 q) 10
n5 q       = div (r10 q) 5
r5 q       = mod (r10 q) 5
n1 q       = div (r10 q) 5
```

Supondo que não queremos generalizar todas as funções menores ainda poderíamos escrever o programa usando definições locais.

```
ncedulas q = n100 + n50 + n10 + n5 + n1
           where
           n100 = div q 100
           r100 = mod q 100
           n50 = div r100 50
           r50 = mod r100 50
           n10 = div r50 10
           r10 = mod r50 10
           n5 = div r10 5
           n1 = mod r10 5
```

Podemos ainda considerar que se houver uma troca no sistema, de modo a incluir uma nova cédula que não seja múltiplo de seus valores menores. Seria fácil mudar o programa para contemplar a mudança nas leis do mundo do problema.

3.5. PROVÉRBIOS

O professor Polya também nos sugere que a lembrança de alguns provérbios pode ajudar o aprendiz (e o resolvedor de problemas) a organizar o seu trabalho. Diz Polya que, apesar dos provérbios não se constituírem em fonte de sabedoria universalmente aplicável, seria uma pena desprezar a descrição pitoresca dos métodos heurísticos que apresentam.

Alguns são de ordem geral, tais como:

O fim indica os meios.

Seus melhores amigos são O que, Por que, Onde, Quando e Como. pergunte O que, pergunte Por que, pergunte Onde, pergunte Quando e pergunte Como - e não pergunte a ninguém quando precisar de conselho.

Não confie em coisa alguma, mas só duvide daquilo que merecer dúvida.

Olhe em torno quando encontrar o primeiro cogumelo ou fizer a primeira descoberta; ambos surgem em grupos.

A seguir apresentamos uma lista deles, organizados pelas etapas às quais parecem mais relacionados.

Etapa 1 : Compreensão do problema.

Quem entende mal, mal responde.

Pense no fim antes de começar.

O tolo olha para o começo, o sábio vê o fim.

O sábio começa pelo fim, o tolo termina no começo.

Etapa 2 : Planejamento da solução.

A perseverança é a mãe da boa sorte.

Não se derruba um carvalho com uma só machadada.

Se no princípio não conseguir, continue tentando.

Experimente todas as chaves do molho.

Veleja-se conforme o vento.

Façamos como pudermos se não pudermos fazer como queremos.

O sábio muda de opinião, o tolo nunca.

Mantenha duas cordas para um arco.

Faça e refaça que o dia é bastante longo.

O objetivo da pescaria não é lançar o anzol mas sim pegar o peixe.

O sábio cria mais oportunidades do que as encontra.

O sábio faz ferramentas daquilo que lhe cai às mãos.

Fique sempre de olho na grande ocasião.

Etapa 3: Construção da solução.

Olhe antes de saltar.

Prove antes de confiar.

Uma demora prudente torna o caminho seguro.

Quem quiser navegar sem risco, não se faça ao mar.

Faça o que puder e espere pelo melhor.

É fácil acreditar naquilo que se deseja.

Degrau a degrau sobe-se a escada.

O que o tolo faz no fim, o sábio faz no princípio.

Etapa 4: Avaliação da solução.

Não pensa bem quem não repensa.

É mais seguro ancorar com dois ferros.

Exercícios

1) Compare as duas definições para a função que descreve o número mínimo de cédulas para pagar uma quantia q, ncedulas e ncedulas2. Discuta;
2) Desenvolva a solução para o problema da cédulas considerando o fato de que o Real possui notas de 2 e notas de 20. O que muda?
3) Apresente três problemas semelhante ao das cédulas;
4) Desenvolva uma solução para o problema considerando que para cada tipo de cédula existe um quantidade limitada (maior ou igual a zero);

4. ABSTRAÇÃO, GENERALIZAÇÃO, INSTANCIAÇÃO E MODULARIZAÇÃO

4.1. INTRODUÇÃO

Na busca por resolver um problema podemos usar vários princípios, cada um evidentemente terá uma utilidade para a resolução do problema, ou para garantia de sua correção ou ainda para facilitar os usos posteriores da solução que obtivermos. Apresentamos a seguir alguns deles.

4.2. ABSTRAÇÃO

Quando escrevemos uma expressão e não damos nome a ela, o seu uso fica limitado àquele instante específico. Por exemplo, suponha que desejamos determinar a hipotenusa de um triângulo retângulo com catetos 10 e 4. Como conhecemos o teorema de Pitágoras ($a^2 = b^2 + c^2$), podemos usar diretamente nossa máquina funcional para avaliar a seguinte expressão:

```
> sqrt ((10.0 * 10.0)+ (4.0 * 4.0))
10.7703
```

A expressão que codificamos serve apenas para esta vez. Se em algum outro instante precisarmos avaliá-la, teremos que codificá-la novamente.

Para evitar isso, é que damos nomes às nossas expressões, para que possamos usá-las repetidamente, apenas referenciando-as pelo seu nome. No caso acima, poderíamos escrever a definição:

hipotenusa = sqrt ((10.0 * 10.0)+ (4.0 * 4.0))

De posse dessa definição nossa máquina poderá avaliar a expressão sempre que dela precisarmos. Basta escrevê-la:

```
> hipotenusa
10.7703
```

Você pode agora estar se perguntando, porque não trocamos a definição para usar diretamente o valor ***10.7703***?

hipotenusa = *10.7703*

Agindo assim a máquina não precisaria avaliar a expressão ***sqrt ((10.0 * 10.0)+ (4.0 * 4.0))*** a cada uso. Por outro lado não ficaria registrada a origem do valor ***10.7703***, com o tempo perderíamos esta informação. De qualquer forma, teríamos criado uma abstração à qual denominamos ***hipotenusa***.

Outra pergunta pode estar rondando a sua cabeça, por que escrever uma definição que é sempre avaliada para o mesmo valor, por que não generalizá-la? Bom, este foi apenas um recurso didático para separar os dois conceitos: a abstração que acabamos de apresentar e a generalização que apresentaremos a seguir. No entanto convém lembrar que algumas definições são realmente constantes e que são de grande utilidade, como é o caso da seguinte definição:

pi = 3.1416

4.3. GENERALIZAÇÃO

Quando uma abstração se aplica a vários valores podemos generalizá-la. Assim, além de usá-la várias vezes para os mesmos valores, podemos também usá-la para valores diferentes. Esta ultima alternativa evidentemente facilita o seu reuso.

Vamos apresentar duas formas para fazer isso:

a) Elaborando a definição usando outras definições constantes. Por exemplo, no caso da definição acima para a hipotenusa, poderíamos escrever as definições a seguir:

b = 10.0
c = 4.0
hipotenusa = sqrt ((b * b) + (c * c))

Aqui, ***hipo*** se aplica a dois valores quaisquer ***b*** e ***c***, os quais são objetos de outras definições.

b)Uma forma mais geral é através do conceito de parametrização. Esta consiste em indicar no cabeçalho da definição (lado esquerdo da igualdade) quais são os objetos da generalização. Para o exemplo que estamos trabalhando, podemos escrever:

hipotenusa b c = sqrt ((b * b) + (c * c))

Temos então descrito a função paramétrica ***hipotenusa*** cujos parâmetros são ***b*** e ***c***.

4.4 INSTANCIAÇÃO

A parametrização permite que usemos a mesma definição para diferentes instâncias do problema. Por exemplo, suponha que desejamos determinar a hipotenusa de três triângulos retângulos. O primeiro com catetos 10 e 4, o segundo com catetos 35 e 18 e o terceiro com catetos 9 e 12. Podemos instanciar a definição paramétrica para estabelecer a seguinte interação:

> hipo 10.0 4.0
10.7703
> hipo 35.0 18.0
39.3573
> hipo 9.0 12.0
15.0

4.5. MODULARIZAÇÃO

Em geral, nossos problemas não serão tão simples e diretos quanto o exemplo acima. Quando nos deparamos com problemas maiores, um bom princípio é:

Divida para facilitar a conquista.

Basicamente este princípio consiste em quebrar o problema inicial em problemas menores, elaborar a solução para cada um dos problemas menores e depois combiná-las para obter a solução do problema inicial. A cada um dos subproblemas encontrados

podemos reaplicar o mesmo princípio. Segundo Polya, esta é uma heurística muito importante à qual ele denomina de Decomposição e Combinação. Antes de pensar em codificar a solução em uma linguagem de programação específica, podemos representá-la através de um esquema gráfico denominado de estrutura modular do problema (veja a representação para o exemplo a seguir).

4.5.1 MODULARIZANDO: Considere o problema de descrever a área da Figura 4.1 abaixo.

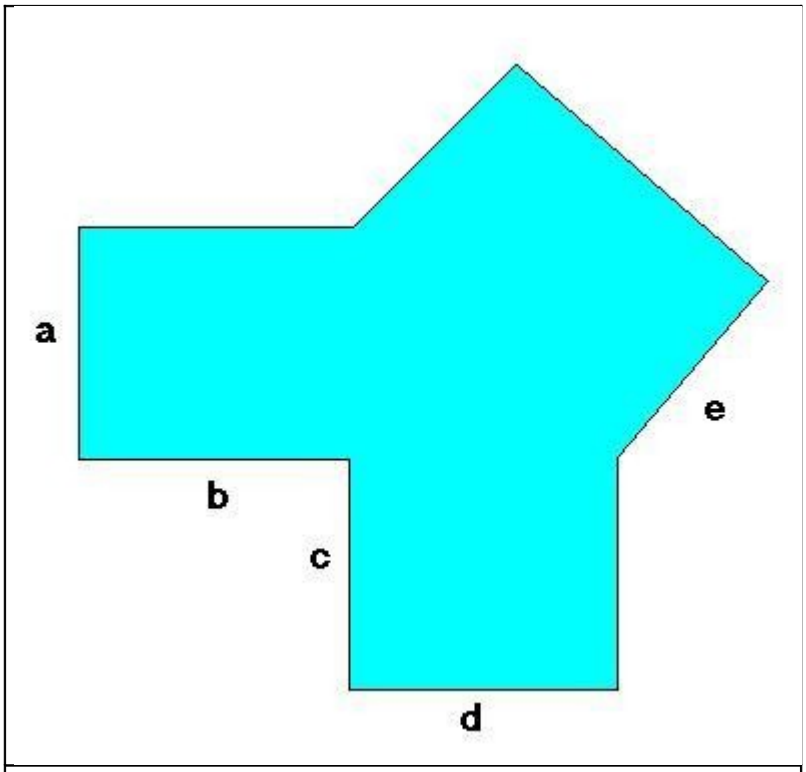
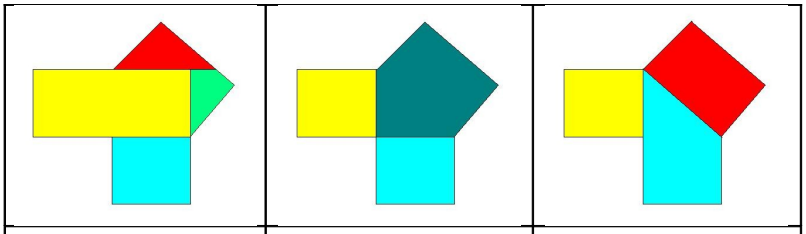


Figura 4.1 Um polígono irregular para o qual desejamos determinar a sua área

Como podemos concluir por uma inspeção da figura, não existe uma fórmula pronta para calcular sua área. Precisamos dividir a figura em partes para as quais conheçamos uma maneira de calcular a área e que, além disso, conheçamos as dimensões necessárias.

Podemos fazer várias tentativas, como por exemplo, as ilustradas nas Figuras 4.2 a, 4.2 b e 4.2 c.



a b c

Figura 4.2 – Possíveis subdivisões do polígono apresentado na Figura 4.1.

Podemos tentar descrever a área das figuras menores em cada uma das figuras apresentadas. Nas Figuras 4.2.b e 4.2c, parece que precisaremos subdividir novamente. Em 4.2 b a "casinha" pode ser transformada em um retângulo e um triângulo. Em 4.2 c é a "casinha" azul (meia-água) que pode ser dividida em um retângulo e um triângulo, como na Figura 4.3. E a Figura 4 a? Que podemos dizer?

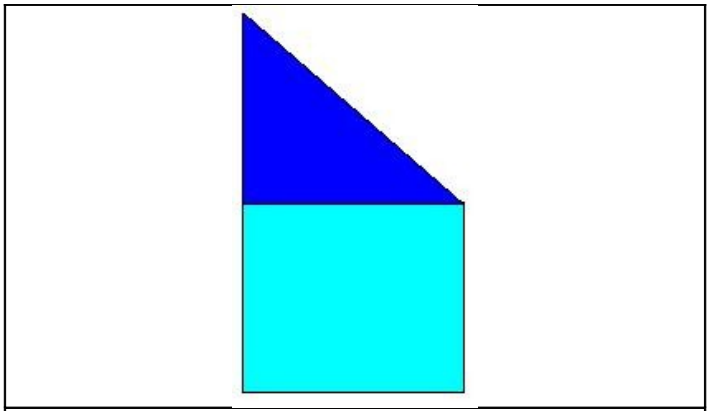


Figura 4.3 – Subdivisão de uma sub-figura da Figura 4.3 c,

Vamos partir para a nossa solução a partir da figura Fig. 4.2c. Podemos dizer que a área total pode ser obtida pela soma das áreas amarela, vermelha e azul. A área azul pode ser subdividida em azul-claro e azul-escuro (ver Figura 4.3).

área total	=	área amarela + área vermelha + área azul
área azul	=	área azulclaro + área azulescuro

Quando escolhemos as áreas acima citadas, não foi por acaso. A escolha foi baseada na simplicidade do subproblema. Podemos usar este conhecimento para chegar a um outro. Três das áreas que precisamos calcular são de forma retangular. Ou seja, são especializações de um conceito mais geral. A quarta área, também pode ser obtida pela especialização do conceito de triângulo retângulo.

Vamos agora aproximar nossas definições da linguagem de programação.

Portanto podemos escrever:

a_retangulo x y = x * y

O triângulo que temos é retângulo e podemos descrever sua área por:

a_t_retangulo x y = (x * y) / 2.0

A determinação da área vermelha nos traz uma pequena dificuldade. Não nos foi informado qual o comprimento da base do retângulo. E agora? Observando bem a figura podemos concluir que a base do retângulo é igual à hipotenusa do triângulo retângulo de catetos ***a*** e ***d.*** Podemos escrever então:

atotal a b c d e	=	a_retangulo a b + a_retangulo (hipo a d) e + a_azul a c d
a_azul x y z	=	a_retangulo y z + a_t_retangulo x y
hipo x y	=	sqrt (x * x + y * y)
a_retangulo x y	=	x * y
a_t_retangulo x y	=	(x * y) / 2.0

A estrutura da solução de um problema obtida pela modularização pode ser representada por um diagrama denominado árvore. Para o problema acima discutido a árvore é da forma apresentada na Figura 4.5.

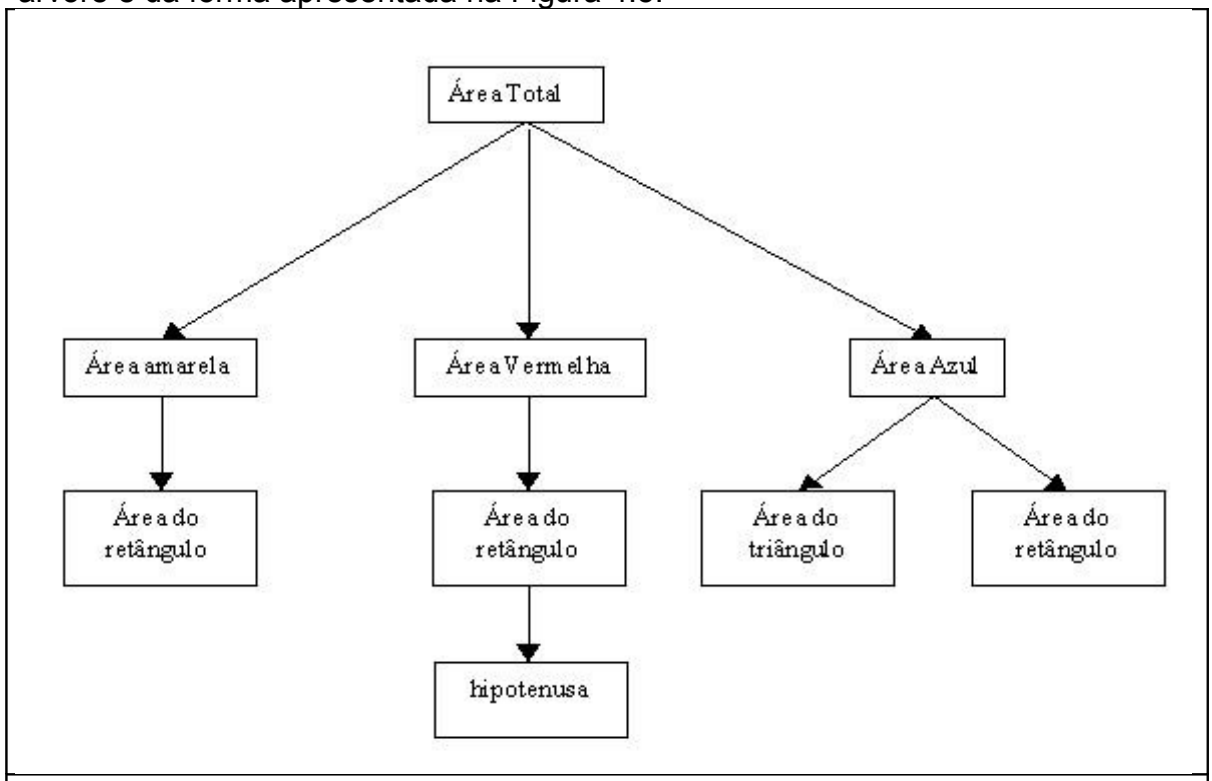


Figura 4.5 – Representação em árvore da estrutura modular do problema “cálculo da área do polígono irregular apresentado na Figura 4.1”.

4.6. UM EXEMPLO DETALHADO

Problema: Escreva uma descrição funcional para o volume de cada uma das peças apresentadas nas Figuras 4.6 e 4.7.

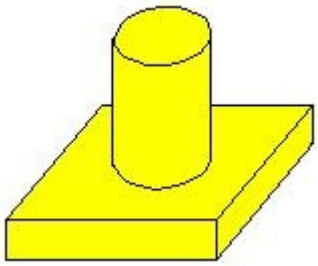


Figura 4.6 – Peça no. 1

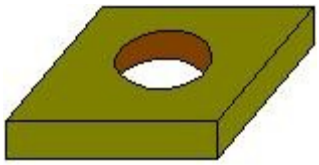


Figura 4.7 – Peça no. 2

Solução 1: Vamos começar pela Figura 4.6. Uma rápida análise nos leva a identificar duas partes. Na parte inferior temos um paralelepípedo, sobre o qual se assenta um cilindro. Vamos supor que são maciços. Chamemos de *a*, *b* e *c* as dimensões do paralelepípedo, de *r* o raio da base do cilindro e de *h* a sua altura. O volume total da peça pode ser determinado pela soma do volume das duas peças menores. Chamemos a função de volfig46. Uma possível definição para essa função é:

volfig46 a b c r h = (a * b * c) + (3.1416 * r * r * h)

Vamos então tratar da peça descrita na figura 4.7. Agora identificamos uma peça apenas. Um paralelepípedo com um furo no centro. Chamemos de *a*, *b* e *c* as dimensões do paralelepípedo, de *r* o raio do buraco. Para descrever o volume da peça devemos subtrair do volume do paralelepípedo o volume correspondente ao buraco. Chamemos a função de volfig47. Uma possível definição para volfig47 é:.

volfig47 a b c r = (a * b * c) - (3.1416 * r * r * c)

Solução 2: A solução 1, apesar de resolver o problema, deixa de contemplar algumas práticas importantes. No tratamento da primeira peça (Figura 4.6), apesar de identificadas duas peças menores, estas abstrações não foram descritas separadamente. Ou seja, deixamos de registrar formalmente a existência de duas abstrações e com isso não pudemos modularizar a descrição da função principal. Podemos tentar então um outro caminho, contemplando as abstrações para o cilindro e para o paralelepípedo:

volcil r h	= 3.1416 * r * r * h
volpar a b c	= a * b * c
Volfig46 a b c r h	= volpar a b c + volcil r h

Voltemos então para a segunda peça (Figura 4.7), e vejamos se podemos identificar similaridades. Aqui só temos uma peça, que se parece com o paralelepípedo da primeira figura. Como podemos identificar similaridades? Aprofundando melhor nossa análise podemos lembrar que o furo no paralelepípedo, corresponde a um cilindro que foi retirado. Desta forma, podemos então concluir que o volume da figura 4.7 pode ser obtido pela diferença entre o volume de um paralelepípedo e de um cilindro. Como já temos as definições para volume de cilindro e volume de paralelepípedo, só nos resta escrever a definição final do volume da figura 4.7:

volfig47 a b c r = volpar a b c - volcil r c

Analisando a solução, podemos tirar algumas conclusões:

a) a solução ficou mais clara,
b) a solução propiciou o reaproveitamento de definições.
c) se precisarmos usar volume de cilindro e de paralelepípedo isoladamente ou em outras combinações, já as temos disponíveis.

Considerações complementares - As descrições das áreas de um retângulo e de uma circunferência podem ser dadas respectivamente por:

```
acir r   = pi * r * r
aret a b = a * b
```

Desta forma, podemos reescrever o volume do cilindro e do paralelepípedo da seguinte maneira:

```
volcil r h = acir r * h
volpar a b c = aret a b * c
```

Solução 3: Se aprofundarmos a análise podemos observar que as duas figuras podem ser abstraídas como uma só! O cilindro que aparece na primeira, como um volume que deve ser acrescentado pode ser entendido como o mesmo cilindro que deve ser subtraído na segunda. Além disso, podemos estender a solução para furos que não vazem a peça, pois a altura do cilindro pode ser qualquer. Considere a definição a seguir:

```
volfig a b c r h = volpar a b c + volcil r h
```

Podemos observar que esta é a mesma definição apresentada anteriormente para a Figura 4.6. O que muda é o uso. Para calcular o volume da Figura 4.6 usamos um ***h*** positivo e para Figura 4.7 um ***h*** negativo. Observe os exemplos a seguir:

> volfig 6.0 8.0 2.0 2.0 5.0 158.832	-- determina o volume de uma instância da figura Fig. 4.6, com um cilindro de raio 2 e altura 5.
> volfig 6.0 8.0 2.0 2.0 (-2.0) 70.8673	-- determina o volume de uma instância da figura Fig. 4.7, vasada por um furo de raio 2. -- neste caso fornecemos um valor negativo para a altura do cilindro com o mesmo valor da altura do paralelepípedo
> volfig 6.0 8.0 2.0 2.0 (-1.0) 83.4336	-- determina o volume de uma instância da figura Fig. 4.7, que tem um furo de raio 2 e profundidade 1.

Exercícios:

1) Discuta o significado da expressão **volfig 6.0 8.0 2.0 2.0 (-5.0)**. O que poderia ser feito para contornar este efeito indesejável?
2) Resolva o problema da Figura 4.1 usando a modularização sugerida pela figura 4.2 b.
3) Redesenhe a Figura 4.5 (árvore de modularização) para a solução apresentada ao exercício 2.
4) Apresente 3 problemas similares para cálculo de área de polígono irregular.
5) Apresente 3 problemas similares para cálculo de volume de objetos.
6) Apresente 3 problemas similares em outros domínios de conhecimento.

5. TIPOS DE DADOS NUMÉRICOS

5.1. INTRODUÇÃO

Denominamos **Tipo de Dados** a um conjunto de valores, munido de um conjunto de operações sobre esses valores. Por exemplo, podemos denominar de T1 ao tipo de dados formado por um conjunto S de valores idêntico aos números naturais (S = {0,1,2,3, ...}) e munido das operações de adição (***a***) e multiplicação (***m***). Cada operação possui, por sua vez, um tipo, indicando o domínio e o contradomínio. Por exemplo, para o tipo T1, o domínio de ***a*** é ***S X S e o*** contradomínio é ***S.*** Como visto anteriormente, a notação a seguir é usualmente utilizada e em geral é denominada de "assinatura" da operação.

a :: S X S → S
m :: S X S → S

Como visto no Capítulo 2, na notação Curry escreveríamos:

a :: S → S → S
m :: S → S → S

Em Haskell, conhecendo-se o tipo das operações e funções que compõem uma expressão podemos determinar o tipo do valor que dela resultará, ou seja, o seu contra-domínio. Em linguagens de programação isto equivale a dizer que a linguagem é ***fortemente tipada***. Dizemos ainda que os tipos são elementares ou estruturados. Os numéricos são elementares, assim como também o são os tipos lógico e os caracteres. Neste capítulo trataremos dos tipos numéricos, os demais virão em capítulos posteriores.

Os números formam um tipo de dados fundamental na computação. Aqui nos interessa subdividi-los em inteiros e reais. Antes de irmos além, é importante que se saiba que, sendo o computador composto de elementos finitos, algumas adaptações e simplificações precisam ser feitas para o tratamento de números. Em HUGS, quando estamos submetendo expressões para avaliação, podemos desejar que o tipo do resultado seja exibido. Para que isso ocorra podemos usar o comando ***:set***, que ativa ou desativa parâmetros do ambiente. Para ativar um parâmetro usamos uma letra correspondente ao parâmetro, precedido do símbolo "+". Para desativar usamos o símbolo "-". Para saber sobre outros parametros, experimente submeter apenas o comando "***:set***". A seqüência de interações a seguir ilustra a ativação e desativação da exibição dos tipos dos resultados das avaliações.

```
> :set +t
> 2^20
1048576 :: Integer

> :set -t
> 2^100
1267650600228229401496703205376
```

Os parâmetros do interpretador, ativados pelo comando :set, são válidos apenas enquanto a sessão do HUGS estiver ativa.

5.2. NÚMEROS INTEIROS

Para trabalhar com números inteiros, a linguagem Haskell provê o tipo **Integer**, que pode produzir números com uma quantidade ilimitada de algarismos. Entretanto, como a memória do computador é finita, qualquer que seja a máquina real que estivermos usando, inevitavelmente esbarrará em limites. Na ilustração a seguir podemos observar essa flexibilidade, quando obtemos o valor para a expressão 2^{1000}, um número de 302 algarismos. Claro, o limite pode estar bem longe e podemos não atingí-lo em nossas aplicações.

```
> 2^1000
10715086071862673209484250490600018105614048117055336074437503883703510511
12493612249319837881569585812759467291755314682518714528569231404359845775
57469857480393456777482423098542107460506237114187795418215304647498358194
41267398767559165543946077062914571196477686542167660429831652624386837205
5668069376
```

Experimente com números maiores! Por exemplo, 9999^{9999} é um número que tem por volta de 40 mil algarismos. É provida ainda uma representação de inteiros mais restrita, denominada ***Int***. Esta versão trabalha com um intervalo de valores fixo e reduzido e tem como vantagem economizar memória do computador e tempo de processamento, usando características específicas do computador. Para que um número seja representado nesta versão, devemos indicar explicitamente, como no exemplo a seguir.

```
> 1234567890::Int
1234567890 :: Int

> 12345678901::Int
Program error: arithmetic overflow
```

O exemplo a seguir ilustra a diferença entres as duas possibilidades.

```
> 1089979879870979879
1089979879870979879 :: Integer

> 1089979879870979879::Int
Program error: arithmetic overflow
```

O conjunto de operações associadas ao domínio pode variar. Em geral são fornecidas as seguintes operações primitivas:

Nome	Descrição	Exemplos
+	Adição	> 13 + 15 + 21 + 24 + 27 + 31 131 :: Integer
*	Multiplicação	> 20 * 10 * 98 19600 :: Integer
-	Subtração	> 1234 - 4321 -3087 :: Integer

div,quot	divisão inteira	> div 12834 10 1283 :: Integer
^	Potência	> 2^20 1048576 :: Integer
rem	resto da divisão inteira entre dois inteiros	> rem 12834 10 4 :: Integer > rem (12834 10) -4 :: Integer
mod	módulo da divisão inteira entre dois inteiros	> mod 12834 10 4 :: Integer > mod (-12834) 10 6 :: Integer
abs	valor absoluto	> abs 123 123 :: Integer > abs (-123) 123 :: Integer
signum	produz -1, 0 ou 1, indicando quando o número é negativo, zero ou positivo	> signum (-3888876527829798) -1 :: Integer > signum 0 0 :: Integer > signum 3 1 :: Integer

Algumas observações são importantes:

a) As operações podem ser combinadas para construir expressões mais complexas;

```
> 5 + 12 * 3
41 :: Integer
```

b) As operações possuem precedência, ou seja, existe uma ordem em que devem ser consideradas. No exemplo anterior, a multiplicação (*) tem precedência sobre a adição (+) e portanto é realizada antes da adição. Esta precedência pode ser modificada pelo uso de parêntesis.

```
> (5 + 12) * 3
51 :: Integer
```

c) A operação div (divisão) é parcial, ou seja, não se aplica quando o denominador é nulo. Quando submetemos uma expressão com esta característica, o HUGS não avalia a expressão e sinaliza que há um erro.

```
> div 398 0
```
Program error: divide by zero

5.3. NÚMEROS REAIS

O tipo em Haskell para representação dos números reais nos computadores é denominado ***Float***. Para este tipo não falaremos de uma lista de constantes. Ao invés disso falaremos aqui em magnitude e precisão de um número. Como na notação científica. A precisão nos diz quantos algarismos significativos são usados e a magnitude nos diz qual o maior expoente admitido.

Por exemplo, uma determinada implementação pode utilizar 6 algarismos significativos. Nesta, o número 123456.789 seria representado pelo número 123457.0, onde o 6 foi arredondado para 7.

A magnitude permite a representação de números bem pequenos e bem grandes. Por exemplo, um número muito grande como 99999999999999999999999999999 poderia ser representado por 1.0e+29 e um número bem pequeno como 0.000000000099999999999999999999999999999, poderia ser representado por 1.0e-010. Veja os exemplos a seguir:

> 0.12345678901234567890123456789012345678901234567890
0.123456789012346 **:: Double**

A representação científica é utilizada quando necessário:

> 123456789012345678901234567 8.9
1.23456789012346e+027 **:: Double**

A implementação corrente do HUGS usa uma versão estendida do tipo Float, denominada **Double**. Se desejarmos, podemos representar os números usando uma quantidade menor de algarismos significativos, para isso precisamos explicitar que o número considerado deve ser representado como do tipo **Float**.

> 0.12345678901234567890123456789012345678901234567890**::Float**
0.1234568 **:: Float**

O número pi é definido com uma constante do tipo Double. Se houver necessidade do uso do valor de pi nas expressões descritas no Haskell, basta escrever a palavra pi, que será avaliada como a constante 3,14159... Veja o exemplo a seguir:

> pi
3.14159265358979 :: Double

O conjunto de operações aqui também pode variar. Em geral, as listadas no quadro abaixo são providas.

Nome	Descrição	Exemplos
+	Adição	> 2.5 + 3.3216 + 0.389458 6.211058 :: Double
*	Multiplicação	> 3.2 * 1.23 * 0.208 0.818688 :: Double
-	Subtração	> 3.456789 - 1.344499089877 2.112289910123 :: Double
/	Divisão	> 9.345988 / 234569.34 3.98431781408431e-005 :: Double
^	potência (o expoente tem que ser Int e positivo)	> 1.5324^10 71.4038177956654 :: Double
sin	Seno	> sin pi 1.22460635382238e-016 :: Doublé
cos	Coseno	> cos pi -1.0 :: Double
tan	Tangente	> tan (pi / 4.0) 1.0 :: Double
sqrt	raiz quadrada	> sqrt 8.5 2.91547594742265 :: Double
log	logaritmo na base ***e***	> log 2.71828182845905 1.0 :: Double
logBase	logaritmo na base escolhida	>logBase 10 2 0.301029995663981 :: Double
exp	potência na base ***e***	> exp 1.0 2.71828182845905 :: Double

Da mesma forma como ocorre nos números inteiros, nos reais também é possível combinar operações para construir expressões mais complexas. Também vale para os reais a precedência de operadores. Como nos inteiros podemos usar parêntesis para controlar a ordem em que as operações serão realizadas.

5.4. CONVERSÃO DE TIPOS

Em Haskell os inteiros são tratados como um subconjunto dos reais. Assim, quando temos números reais misturados com números inteiros em uma mesma expressão, os números inteiros são automaticamente tratados como reais.

```
> 3 + 5
8 :: Integer

> 3 + 5.0
8.0 :: Double
```

Existem funções específicas para conversão de tipos:

a) a função **truncate** converte um real x para o menor inteiro menor ou igual x.

```
> truncate pi
3 :: Integer
```

b) a função **round** converte um real x para o inteiro mais próximo de x, ou seja:

round x = truncate (x + 0.5)

```
> round pi
3 :: Integer

> round (exp 1)
3 :: Integer
```

Outras exemplos de funções que existem só com o intuito de conversão de tipos são fromInteger e fromRational. As assinaturas delas são:

fromInteger :: (Num a) => Integer -> a

fromRational :: (Fractional a) => Rational -> a

5.5. PRECEDÊNCIA DOS OPERADORES

Quando aparecem vários operadores juntos na mesma expressão, certas regras de precedência são estabelecidas para resolver possíveis ambigüidades. A ordem em que os operadores são considerados é a seguinte:

1ª) *div*, *mod*, *abs*, *sqrt* e qualquer outra função

2ª) ^

3ª) * /

4ª) +, -

Da mesma forma que na matemática usual podemos usar os parêntesis para forçar uma ordem diferente de avaliação.

Exemplos:

```
> 2 + 3 * 5
17
```

O operador * tem precedência sobre o operador +. Podemos escrever a expressão a seguir para denotar a seqüência de avaliações:

2 + 3 * 5 ➔ 2 + 15 ➔ 17

> (2 + 3) * 5
25

A ordem de avaliação foi alterada pelo uso dos parêntesis. A seqüência de avaliação é portanto:

(2 + 3) * 5 ➔ 5 * 5 ➔ 25

> 3 * mod 10 4 + 5
11

A prioridade é para avaliação do operador mod, seguida da avaliação do operador * e finalmente do operador +. Podemos escrever a seguinte seqüência de avaliação:

3 * mod 10 4 + 5 ➔ 3 * 2 + 5 ➔ 6 + 5 ➔ 11

> 3 ^ mod 10 4
9

A primeira avaliação é do operador mod e em seguida é avaliado o operador ^. A seqüência de avaliação é:

3 ^ mod 10 4 ➔ 3 ^ 2 ➔ 9

> 4 ^ mod (div 20 4) 2
4

A seqüência de avaliação é:

4 ^ mod (div 20 4) 2 ➔ 4 ^ mod 5 2 ➔ 4 ^ 1 ➔ 1

5.6. ORDEM DE ASSOCIAÇÃO

Quando há ocorrência de operadores de mesma precedência leva-se em consideração a ordem de associação que pode ser à direita ou à esquerda.

1. O operador de subtração faz associação à esquerda,

   5 - 4 - 2 = (5 - 4) - 2 e não 5 - (4 - 2)

2. Já o operador de exponenciação faz associação à direita,

   3 ^ 4 ^ 5 = 3 ^ (4 ^ 5) e não (3 ^ 4) ^5

O uso adequado das noções de precedência e associação serve também para escrevermos expressões com economia de parêntesis.

Observações:

a) O operador unário - deve ser sempre representado entre parênteses quando utilizado junto com outro operador: (- x)^y ou - (x^y) e nunca -x^y ou x^-y

b) A exponenciação, quando repetida em uma expressão, é avaliada da direita para a esquerda: 2^3^3 = 2^(3^3)

c) Os demais operadores, na situação acima, são avaliados da esquerda para a direita: 2 - 3 - 5 = (2 - 3) – 5 e 2 + 3 + 3 = (2 + 3) + 3

Exemplos:

```
>3 - 7 - 2
- 6> 3 * 7 + 4
25

> 3 * ( 7 + 4)
33

> 3 ^ ( 1 + 3)
81
```

5.6. TIPOS DE NOVAS DEFINIÇÕES DE FUNÇÕES

Ao definir novas funções, o fazemos tomando por base os tipos de dados existentes na linguagem. O corpo dessas definições tem por base a composição das operações fornecidas por estes tipos básicos. Assim, o tipo do dado resultante da avaliação de uma aplicação desta nova função, pode ser antecipado por um analisador de tipos, antes mesmo de avaliar a expressão.

Observe as duas definições a seguir:

```
mediaA  x y = (x + y) / 2 e mediaB x y = truncate ((x + y) / 2)
```

O tipo de dado resultante da aplicação da função mediaA **é real, uma vez que a operação "/" (divisão) só se aplica a números reais. Da mesma forma, o tipo resultante da aplicação da função** mediaB **é inteiro, dado que a última operação realizada (**truncate**), converte reais para inteiros. Confira nas avaliações a seguir:**

```
> mediaA 3 4
3.5 :: Double

> mediaB 3 4
3 :: Integer
```

5.7. HIERARQUIA DE TIPOS

Até agora dissemos que a linguagem Haskell trabalha com dois tipos numéricos, os reais e os inteiros. Isto é uma verdade irrefutável. Ao avaliar uma expressão numérica o resultado será sempre um número real ou um número inteiro.
Entretanto, para viabilizar as conversões automáticas de tipo, a linguagem Haskell realiza os números através de uma hieraquia de classes. A classe numérica mais geral é **Num**, dela derivam as classes **Real** e **Fractional**. De **Real** derivam **Integral** e **RealFrac**. De **Fractional** derivam **RealFrac** e **Floating**. De **RealFrac** e de **Floating** deriva **RealFloating**. Algumas classes derivam de mais de uma classe, ao que é denominado de herança múltipla, como é o caso de **RealFrac** que deriva das classes **Real** e **Fractional**. A Figura 5.1 apresenta a hierarquia completa. Podemos observar aqui três classes de apoio, usadas para estruturar as derivações, **Ord**, **Eq** e **Enum**.

Aqui devemos entender classe como uma coleção de operadores sobre um determinado conjunto de valores. Uma classe derivada herda todas as funções providas por sua ancestral e pode acrescentar novas.

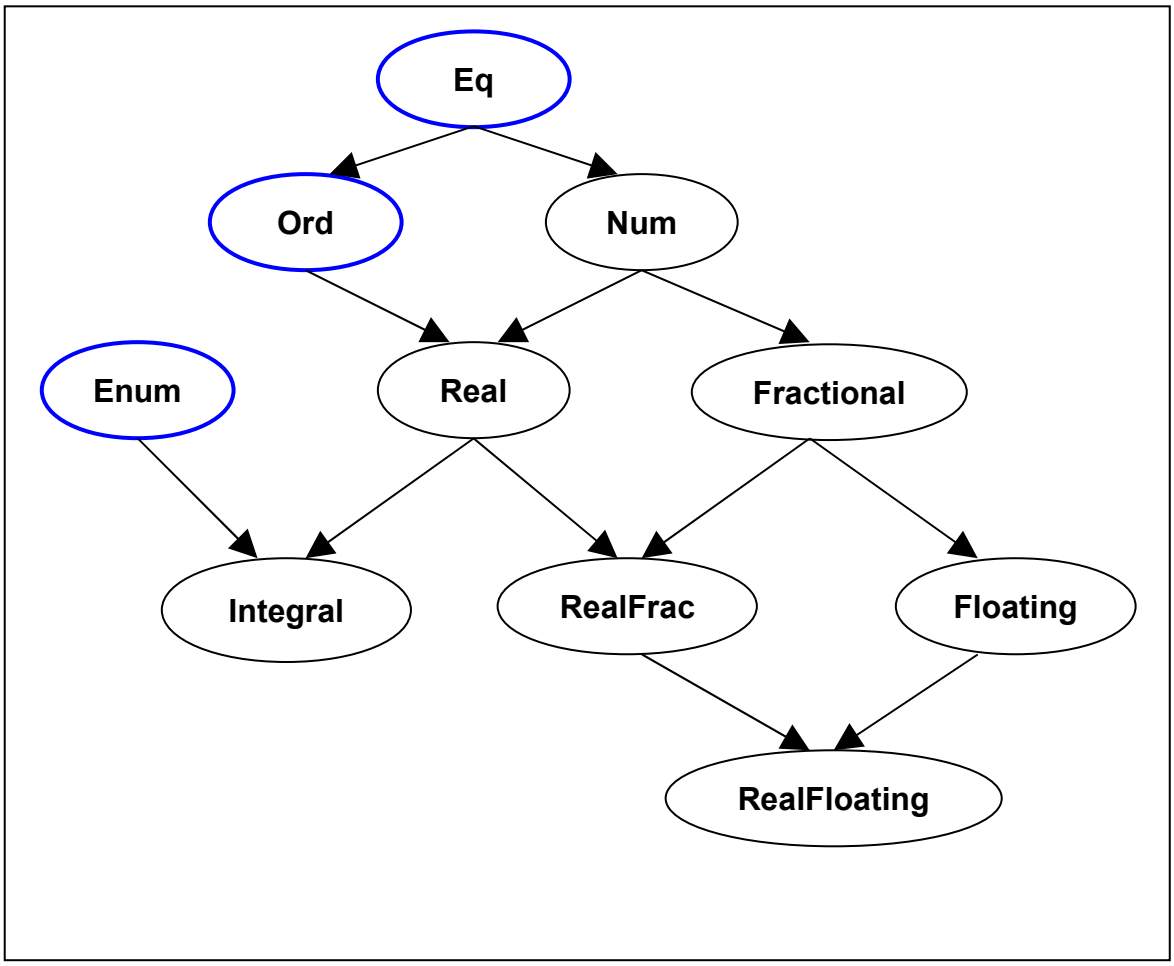


Figura 5.1 – Hierarquia de Tipos

5.7.1 Eq

A classe Eq introduz na linguagem a possibilidade de fazer-se distinção entre os valores de um determinado tipo, sendo a classe mais básica. Com valores que são instâncias desta classe podemos aplicar as operações de igualdade e desigualdade. Podemos pensar como se esta classe tivesse como membros todos os valores sobre os quais se pode fazer distinção. O tipo Bool será apresentado em detalhes no Capítulo 6.

funções	assinatura	descrição informal
(==)	a -> a -> Bool	Avalia se dois valores são idênticos
(/=)	a -> a -> Bool	Avalia se dois valores são distintos

Observemos que a comparação é sempre realizada com valores do mesmo tipo.

Exemplos:

```
Hugs> 3 == 3.0
True :: Bool
Hugs> 3.0 /= 3
False :: Bool
Hugs> 'a' == 'a'
True :: Bool
Hugs> 'a' == 'b'
False :: Bool
Hugs> 'a' == head "abacate"
True :: Bool
Hugs> [1,2,3]==[1..3]
True :: Bool

Hugs> [1,2,3]==['1', '2', '3']
ERROR - Cannot infer instance
*** Instance   : Num Char
*** Expression : [1,2,3] == ['1','2','3']

Hugs> 3 /= 'a'
ERROR - Cannot infer instance
*** Instance   : Num Char
*** Expression : 3 /= 'a'

Hugs> 3.0 == 2.99999999999999999999999999999
True :: Bool
Hugs> 2.99999999999999999999999999999 == 2.9999999999999999
True :: Bool
Hugs> 2.99999999999999999999999999999 == 2.999999999999999
False :: Bool
```

Observemos que:

a) A comparação é sempre realizada com valores do mesmo tipo. Nos exemplos acima, a tentativa de comparar valores de tipos diferentes resulta em uma mensagem de erro, avisando que estamos tentando comparar um valor da classe **Num** com um valor da classe **Char**;

b) A igualdade de números é dependente de uma representação interna, como podemos observar pelos três últimos exemplos.

5.7.2 Ord

Esta é uma classe mais geral que as dos números, provendo suporte para construção de outras classes, inclusive as classes numéricas. A idéia geral é prover operações para todos os conjuntos de valores que podem ser ordenados.

funções	assinatura	descrição informal
compare	a -> a -> Ordering	Compara dois valores associando-os a constantes LT, EQ e GT compare x y = ▪ LT se x é menor que y ▪ EQ se x é igual a y ▪ GT se x é maior que y
(<), (<=), (>=), (>)	a -> a -> Bool	Operadores relacionais que compara dois valores resultando em um valor booleano (True ou False)
max	a -> a -> a	max x y = ▪ x se x é maior ou igual a y ▪ y em caso contrário
min	a -> a -> a	min x y = ▪ x se x é menor ou igual a y ▪ y em caso contrário

Exemplos:

```
Hugs> compare 3 4
LT :: Ordering
Hugs> min 3 4
3 :: Integer
Hugs> min 4 3
3 :: Integer
Hugs> max 4 3
4 :: Integer
Hugs> max 3 4
4 :: Integer
Hugs> 3 <=4
True :: Bool
Hugs> compare 'g' 'b'
GT :: Ordering
Hugs> "abacaxi" > "abacate"
True :: Bool
Hugs> max "abacaxi" "melancia"
"melancia" :: [Char]
Hugs> min "abacaxi" "melancia"
```

```
"abacaxi" :: [Char]
```

5.7.3 Enum

Esta é uma classe que introduz funções para descrever e operar com seqüência de valores.

funções	assinatura	descrição informal
succ,	a -> a	
pred	a -> a	
toEnum	Int -> a	
fromEnum	a -> Int	
enumFromThen	a -> a -> [a]	[n,n'..]
enumFromTo	a -> a -> [a]	[n..m]
enumFromThenTo	a -> a -> a -> [a]	[n,n'..m]

Atemos-nos aqui a comentar as funções “succ” e “pred”, as demais serão apresentadas junto com listas, no capitulo que introduz este tipo de valores. A função “**succ**” determina o sucessor de um valor na seqüência de valores da qual faz parte e a função “**pred**” determina o seu predecessor.

Exemplos:

```
Hugs> succ (-5)
-4 :: Integer
Hugs> pred 0
-1 :: Integer
Hugs> succ 0
1 :: Integer
Hugs> pred (succ 0) == succ (pred 0)
True :: Bool
Hugs> pred 'a'
'`' :: Char
Hugs> pred 'b'
'a' :: Char
Hugs> succ 'B'
'C' :: Char
Hugs> succ '0'
'1' :: Char
Hugs> pred 1.0
0.0 :: Double
Hugs> succ 0.99999
1.99999 :: Double
```

5.7.4 Num

A classe **Num** é a mais geral para os números e provê as funções, (+), (-), (*), negate, signum e abs.

funções	assinatura	descrição informal
`(+), (-), (*)`	`a -> a -> a`	Adição, subtração e multiplicação usuais
`negate`	`a -> a`	Define o simétrico de um número
`abs`	`a -> a`	Define o valor absoluto de um número
`signum`	`a -> a`	signum x = ▪ 1 se x é positivo; ▪ -1 se x é negativo e ▪ 0 se x é nulo
`fromInteger`	`Integer -> a`	Converte um valor do tipo **Integer** em um valor do tipo **Num**

Na prática isto significa que qualquer número usado em Haskell pode ser operado por estas funções.

Exemplos:

```
Hugs> 3 + 4
7 :: Integer
Hugs> 3.0 + 4.0
7.0 :: Double
Hugs> :t 3 + 4
3 + 4 :: Num a => a
Hugs> :t 3.0 + 4.0
3.0 + 4.0 :: Fractional a => a
Hugs> 3 + 4
7 :: Integer
Hugs> 3.0 + 4.0
7.0 :: Double
Hugs> negate 3
-3 :: Integer
Hugs> negate 4.0
-4.0 :: Double
Hugs> signum (-3)
-1 :: Integer
```

5.7.5 Real

A classe **Real** possui uma herança múltipla, descendendo das classes **Num** e **Ord**. Isto significa que com um valor desta classe podemos operar tanto com as operações da classe **Num** quanto com as operações providas por **Ord**. A classe Real introduz a função toRational para conversão de um valor do tipo **Real** em um valor do tipo **Rational**.

funções	assinatura	descrição informal
`toRational`	`a -> Rational`	toRational x = p % q ▪ onde p e q são números inteiros e x é o cociente de p dividido por q

Exemplos:

```
Hugs> toRational 5.5
11 % 2 :: Ratio Integer
Hugs> toRational 1.25
5 % 4 :: Ratio Integer
Hugs> toRational 0.75
3 % 4 :: Ratio Integer
Hugs> toRational 75
75 % 1 :: Ratio Integer
Hugs> toRational 1.5
3 % 2 :: Ratio Integer
Hugs> 3%4 + 2%3
ERROR - Undefined variable "%"
```

A classe **Rational** não está disponível no módulo Prelude e precisa ser importada ou carregada da biblioteca (package/haskell98/ratio.hs). No último exemplo acima a tentativa de operar dois valores do tipo **Rational** produz uma mensagem de erro, tendo em vista que a biblioteca não havia sido carregada.

5.7.6 Fractional

A classe **Fractional** descende diretamente da classe **Num** e introduz a divisão, a função inversa e uma conversão de valores do tipo Rational para Fractional.

funções	assinatura	descrição informal
`(/)`	`a -> a -> a`	
`recip`	`a -> a`	recip x = 1 / x
`fromRational`	`Rational -> a`	

Exemplos:

```
Hugs> recip 5
0.2 :: Double
Hugs> toRational 2.5
5 % 2 :: Ratio Integer
Hugs> fromRational (5 % 2)
ERROR - Undefined variable "%"
Hugs> :l "packages/haskell98/ratio.hs"
Ratio> fromRational(3 % 4)
0.75 :: Double
```

Nos três últimos exemplos podemos observar: a) a tentativa de usar a função fromRational provoca um erro tendo em vista que a biblioteca não estava disponível; b) a carga da biblioteca e c) o uso de fromRational.

5.7.7 Integral

A classe **Integral** possui uma herança múltipla, descendendo das classes **Real** e **Enum**. As funções introduzidas por esta classe são apresentadas a seguir.

funções	assinatura	descrição informal
`quot, rem, div, mod`	`a -> a -> a`	
`quotRem, divMod`	`a -> a -> (a,a)`	
`toInteger`	`a -> Integer`	

Exemplos:

```
Hugs> mod (-5) (3)
1 :: Integer
Hugs> rem (-5) (3)
-2 :: Integer
Hugs> divMod (-5) (3)
(-2,1) :: (Integer,Integer)
Hugs> quotRem (-5) (3)
(-1,-2) :: (Integer,Integer)
```

5.7.9 RealFrac

A classe **RealFrac** descende diretamente das classes **Real e Fractional** introduzindo funções

```
class (Real a, Fractional a) => RealFrac a  where
    properFraction   :: (Integral b) => a -> (b,a)
    truncate, round  :: (Integral b) => a -> b
    ceiling, floor   :: (Integral b) => a -> b
```

5.7.10 Floating

A classe **Floating** descende diretamente da classe **Fractional** introduzindo funções importantes entre as quais as de exponenciação, de logaritmo, raiz quadrada e as trigonométricas.

funções	assinatura	descrição informal
`pi`	`a`	
`exp, log, sqrt`	`a -> a`	
`(**), logBase`	`a -> a -> a`	
`sin, cos, tan`	`a -> a`	
`asin, acos, atan`	`a -> a`	

`sinh, cosh, tanh`	`a -> a`	
`asinh, acosh, atanh`	`a -> a`	

Ao definir novas funções, o fazemos tomando por base os tipos de dados existentes na linguagem. O corpo dessas definições tem por base a composição das operações fornecidas por estes tipos básicos.

Exercícios:

1. Qual o tipo resultante da avaliação da definição da função mediaC x y = div (x + y) 2?
2. O que ocorre na avaliação de uma expressão em Haskell que usa a definição da função mediaD x y = div (x + y) 2.0 ?

6. EXPRESSÕES LÓGICAS E O TIPO BOOLEAN

6.1. INTRODUÇÃO

Uma característica fundamental dos agentes racionais é a capacidade de tomar decisões adequadas considerando as condições apresentadas pelo contexto onde está imerso. Uma máquina que sabe apenas fazer contas, ou seja, manusear as operações aritméticas, terá sua utilidade fortemente reduzida e, portanto não despertará tantos interesses práticos. Os computadores que temos hoje são passíveis de serem programados para tomada de decisão, ou seja, é possível escolher entre duas ou mais ações, aquela que se deseja aplicar em um determinado instante. Em nosso paradigma de programação precisamos de dois elementos fundamentais: um tipo de dados para representar a satisfação ou não de uma condição e um mecanismo que use essas condições na escolha de uma definição. Neste capítulo discutiremos a natureza das proposições lógicas, sua aplicabilidade na resolução de problemas e introduziremos um novo tipo de dados, denominado **boolean**. Satisfazendo logo de saída a curiosidade do leitor lembramos que o nome é uma homenagem a George Boole que estudou e formalizou as operações com estes tipos de valores.

6.2. PROPOSIÇÕES LÓGICAS

Revirando o baú das coisas estudadas no ensino fundamental, por certo encontraremos as sentenças matemáticas. Lembraremos então que elas são afirmações sobre elementos matemáticos, tais como os exemplos a seguir:

1. vinte e dois é maior que cinco
2. dois mais três é igual a oito
3. todo número primo é impar

O conceito de proposição lógica é mais geral, aplicando-se às mais diversas situações do cotidiano, como nos exemplo a seguir:

1. Maria é namorada de Pedro
2. José é apaixonado por Maria
3. Hoje é domingo

Analisando o significado da informação contida em uma proposição lógica, podemos concluir que ela será uma afirmação verídica quando se referir a fatos que realmente acontecem em um determinado mundo. Quando isso não ocorre, concluímos que elas são falsas. Para podermos avaliar uma dada proposição, é necessário ter acesso ao mundo considerado.

Sentenças Fechadas: as sentenças 1 a 6 possuem uma característica importante: todos os seus componentes estão devidamente explicitados. Denominamos estas sentenças de sentenças fechadas. Uma sentença fechada pode ser avaliada imediatamente, conferindo o que elas afirmam com o mundo sobre o qual elas se referem.

Sentenças Abertas: são outro tipo de proposição importante. Nestas, alguns personagens não estão devidamente explicitados e, portanto a sentença não pode ser avaliada. Quando

tratamos sentenças abertas, antes é necessário instanciá-las para algum valor. Por exemplo, a sentença matemática

$$x + 5 > 10$$

nem sempre é verdadeira. Depende do valor que atribuirmos à variável ***x***. Quando atribuímos valores às variáveis de uma sentença dizemos que estamos instanciando a sentença.

No caso acima, podemos instanciar a sentença para os valores 3 e 10, entre outros, obtendo as seguintes instâncias:

$$3 + 5 > 10$$
$$10 + 5 > 10$$

Agora podemos avaliá-las e concluir que a segunda é verídica e a primeira não.

Sentenças Compostas: O discurso do cotidiano e até o discurso matemático podem ser escritos como uma lista de proposições simples. Exemplos:

1. Hoje é domingo
2. Aos domingos tem futebol
3. Quando meu time joga, eu assisto

Nem sempre isto é desejável ou suficiente. Para resolver a questão, podemos contar com as sentenças compostas. Como por exemplo:

a) *Três é menor que cinco e o quatro também;*
b) *Domingo irei ao futebol ou escreverei notas de aula;*
c) *Esperanto não é uma linguagem de programação.*

Observamos então, o surgimento de três palavras para criar essas sentenças e que essas palavras (***e***, ***ou***, ***não***) não se referem a coisas, propriedades ou fenômenos de um determinado universo. Elas são denominadas de palavras lógicas, visto que a sua função na linguagem é possibilitar a articulação entre as proposições do discurso.

A composição de uma proposição pode envolver várias palavras lógicas, como nos exemplos a seguir:

1. ***Dois é menor que três e maior que um mas não é maior que a soma de três com dois;***
2. ***Maria gosta de Pedro e de Joana mas não gosta de Antonio.***

Avaliação de Sentenças Compostas: Para avaliar sentenças simples, vimos que bastava inspecionar o mundo ao qual ela se refere e verificar se a situação expressa pela sentença ocorre ou não. E como proceder para as sentenças compostas? Um caminho que parece confiável é apoiar essa avaliação no papel representado por cada uma das palavras lógicas. Assim, ao considerarmos a proposição

Hoje fui ao cinema e ao teatro,

só poderemos dizer que ela é verídica se tanto a proposição ***Hoje fui ao cinema*** quanto a proposição ***Hoje fui ao teatro*** forem avaliadas como verídicas.

Para a proposição composta

Maria foi à missa ou ao cinema,

devemos considerá-la como verídica se uma das duas proposições:

1. ***Maria foi à missa***
2. ***Maria foi ao cinema***

forem avaliadas como verídica. E se as duas forem avaliadas como verídicas? No discurso cotidiano tendemos a pensar nesta situação como inverídica visto que queríamos uma ou outra. A linguagem natural não estabelece se devemos ou não explicitar que não estamos falando do caso em que ambas podem ser constatadas. Podemos assumir, portanto, que a nossa sentença composta usando ***ou*** será avaliada como verídica quando pelo menos uma das duas proposições que a compõe for avaliada com verídica.

No caso da sentença composta usando a palavra lógica ***não***, como em

Hoje não choveu

diremos que ela será avaliada como verídica quando a proposição ***Hoje choveu*** não puder ser constatada e como inverídica no caso oposto.

6.3. O TIPO DE DADOS BOOLEAN

Podemos agora discutir sobre a natureza do valor resultante da avaliação de uma sentença. A avaliação é de natureza funcional, ou seja, dada uma sentença ***s*** ela será mapeada em um de dois valores distintos. Então, o contradomínio de uma avaliação é um conjunto com apenas dois valores, um para associar com avaliações verídicas e outras com as inverídicas. Podemos escolher um nome para esses valores. Historicamente, as linguagens de programação os tem denominado de ***True*** e ***False***. O primeiro para associar com as proposições que podem ser constatadas no mundo considerado e a segunda com as não constatáveis. Para a avaliação das proposições compostas, podemos considerar uma função matemática cujo domínio e contradomínio são definidos como o conjunto que acabamos de definir com as constantes ***True*** e ***False***. Assim

aval :: <sentença> → {True, False}

As proposições compostas podem ser formadas pelas operações lógicas de conjunção, disjunção ou negação. Em Haskell, essas operações são representadas por:

Operação lógica	**Operador lógico (Haskell)**
e	&&
ou	||
não	not

Vamos então formalizar a avaliação de sentenças compostas.

Sejam s1 e s2 duas proposições lógicas:

1. O valor lógico da sentença ***s1 && s2*** é ***True*** se e somente se o valor lógico de ***s1*** é ***True*** e o valor lógico de ***s2*** é ***True*** e é ***False*** em caso contrário;

2. O valor lógico da sentença ***s1 || s2*** é ***False*** se e somente se o valor lógico de ***s1*** é ***False*** e o valor lógico de ***s2*** é ***False*** e é ***True*** em caso contrário:

3. O valor lógico da sentença ***not s1*** é ***True*** se e somente se o valor lógico de ***s1*** é ***False*** e é ***False*** em caso contrário.

Tabela Verdade: Estas definições também podem ser representadas através de uma enumeração de suas associações, formando o que se costuma chamar de ***Tabelas Verdade*** as quais apresentamos a seguir.

s1	s2	s1 && s2
True	True	True
True	False	False
False	True	False
False	False	False

s1	s2	s1 || s2
True	True	True
True	False	True
False	True	True
False	False	False

s1	not s1
True	False
False	True

6.4. OPERADORES RELACIONAIS

Quando falamos de proposições lógicas no discurso matemático, ou melhor, em sentenças matemáticas, usamos termos tais como: “menor do que”, “menor ou igual”, “diferente”, entre outros. Estes termos são fundamentais para a nossa intenção de prover os computadores com a capacidade de decisão. Denominamos estes elementos de **operadores relacionais**, pois estabelecem uma relação de comparação entre valores de um mesmo domínio. O contradomínio deles é do tipo Boolean. A tabela a seguir apresenta esses operadores, suas representações no Haskell, seus significados e exemplos de uso.

operador	*significado*	*exemplo*	*resultado*
==	igualdade	(2 + 3) == (8 – 3)	True
/=	Diferença	5 /= (4 * 2 -3)	False
<	Menor	(2 + 3) < 6	True
<=	Menor ou igual	(2 * 3) <= 6	True
>	Maior	(4 + 2) > (2 * 3)	False
>=	Maior ou igual	(8 – 3 * 2) >= (15 div 3)	False

Podemos usar estes operadores para construir novas definições ou simplesmente, em uma sessão de Hugs na avaliação de uma expressão, como apresentado nos exemplos a seguir.

```
> 5 > 4
True :: Bool

> 4 /= (5 + 3)
True :: Bool

> (mod 5 2) == (rem 5 2)
True :: Bool

> (mod (-5) 2) == (rem (-5) 2)
False :: Bool

>  (div 5 2) <= truncate(5.0 / 2.0)
True :: Bool
```

6.5. EXPRESSÕES E DEFINIÇÕES

Agora que temos um novo tipo de dados, podemos utilizá-lo a nosso serviço, escrevendo expressões de forma tão natural quanto aquela que usamos para escrever expressões aritméticas. Usando essas expressões podemos então construir definições cujo tipo resultante seja booleano. Os ingredientes básicos para construir essas expressões são os operadores relacionais.

Expressões simples: Por exemplo, para construir uma definição que avalie se um número é par, podemos usar a seguinte definição:

```
par x = (mod x 2) == 0
```

Vejamos alguns exemplos de avaliação da função par:

```
> par 5
False
> par 148
True
> par 0
True
> par (truncate ((5 + 2) / 2))
False
```

Outros exemplos de definições:

• Verificar se ***a*** é múltiplo de ***b***	`multiplo a b = (mod a b) == 0`
• Verificar se ***a*** é divisor de ***b***	`divisor a b = multiplo b a`
• Verificar se uma distância ***d*** é igual à diagonal de um quadrado de lado ***a***	`diag d a = (a * sqrt 2.0) == d`
• Verificar se um número é um quadrado perfeito	`quadp n = (sqrt n)^2 == n`
• Verificar se dois números ***a*** e ***b*** são termos consecutivos de uma P.A. de razão ***r***	`spa a b r = (a + r) == b`

Expressões compostas: Podemos usar agora os operadores lógicos para construir expressões compostas. Veja os exemplos a seguir:

• Verificar se 3 números estão em ordem decrescente	`ordc a b c = (a > b) && (b > c)`

• Verificar se um número ***x*** está no intervalo fechado definido por ***a*** e ***b***	`pert x a b = (x >= a) && (x <= b)`
	
	ou
	
	`pert x a b = not ((x < a) || (x > b))`

• Verificar se um determinado ponto do espaço cartesiano está no primeiro quadrante	`pquad x y = (x > 0) && (y > 0)`

• Verificar se 3 números ***a***, ***b*** e ***c***, são lados de um triângulo retângulo	`tret a b c = ((a^2 + b^2) = = c^2) ||`
	`((a^2 + c^2) = = b^2) ||`
	`((b^2 + c^2) = =a^2)`

Quando nossas expressões possuem mais de um operador lógico devemos observar a precedência de um sobre o outro. Por exemplo, na expressão

P || Q && R

as letras P, Q e R são expressões lógicas. O operador && será avaliado primeiro pois tem precedência sobre o ||, portanto a expressão será avaliada para True se P for avaliado para True ou se a subexpressão (Q && R) for avaliada para True.

Podemos modificar esta ordem utilizando parêntesis como nas expressões aritméticas. A expressão acima poderia ser reescrita como

```
(P || Q) && R
```

Agora, para que ela seja avaliada para verdadeira é preciso que R seja avaliada como verdadeira.

Vejamos mais alguns exemplos de definição de função booleana:

• Verificar se x está fora do intervalo definido por a e b e a e b estão em ordem não decrescente	f x a b	=	((x <=a) || (x >= b)) && (a <= b)
• Verificar se x é menor que a ou se x é maior que b e a e b estão em ordem não decrescente	g x a b	=	(x <=a) || (x >= b) && (a <= b)

6.6. RESOLVENDO UM PROBLEMA: Desejamos verificar se um determinado ponto do espaço cartesiano está dentro ou fora de um retângulo paralelo aos eixos, conhecidos o ponto, o canto superior esquerdo e o canto inferior direito do retângulo.

Etapa 1 [Entendendo o problema] O ponto pode estar em qualquer quadrante? E o retângulo, pode estar em qualquer quadrante? Basta ter os dois cantos para definir o retângulo? Em que ordem serão informados os cantos? Como se descreve um canto? Um ponto que esteja sobre um dos lados, está dentro ou fora do retângulo?

Vamos assumir então as seguintes decisões: discutiremos inicialmente apenas sobre pontos e retângulos localizados no primeiro quadrante. A ordem em que os dados serão informados será: primeiro o ponto, depois o canto superior esquerdo e por último o canto inferior direito. Para cada ponto serão informadas as duas coordenadas, primeiro a abscissa e depois a ordenada. Os pontos na borda são considerados pertencentes ao retângulo.

Etapa 2 [Planejando a Solução]: Conheço algum problema parecido? Posso decompor este problema em problemas mais simples? Sei resolver um problema mais geral em que este é um caso particular?

Bom, já nos envolvemos com o problema para verificar se um ponto estava num intervalo linear, este também se refere a intervalo. Verificar se um ponto pertence a uma região qualquer do espaço n-dimensional é mais geral, se eu conhecesse uma definição para este problema geral, bastaria instanciá-la. Será que posso decompor o problema na verificação de dois espaços lineares, um definido pelos lados paralelos ao eixo das ordenadas e outro paralelo ao eixo das abscissas? Se afirmativo, como combino as duas soluções?

Para que um ponto esteja dentro do retângulo é necessário que sua ordenada esteja entre as ordenadas dos dois cantos. Sabemos também que a abscissa precisa estar entre

as abscissas dos dois cantos. Isso basta? Para combinar as soluções percebemos que é suficiente que as duas primeiras estejam satisfeitas.

Etapa 3 [Construindo a Solução] Construindo a solução - Anteriormente já elaboramos a definição de pertinência a um intervalo linear, vamos usá-la aqui.

pert x a b	=	(x>=a) && (x <= b)	pertinência linear
pertsup x y x1 y1 x2 y2	=	(pert x x1 x2) && (pert y y2 y1)	pertinência no plano

E o teste, para que valores interessam testar?

Etapa 4 [Analisando a Solução]: Existem outras maneiras de resolver o problema? Esta solução se aplica as outras dimensões?

Exercícios

1. Avalie as expressões abaixo, explicando como foram reduzidas para o valor final, que é booleano. Se houver erro de avaliação, identifique a causa do erro.

a) 3 < 4 || 5 == 7

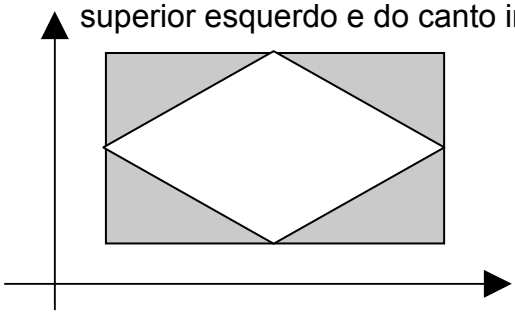
b) not 3 < 7

c) 3+4 <= 8 && 10 /= 34.7

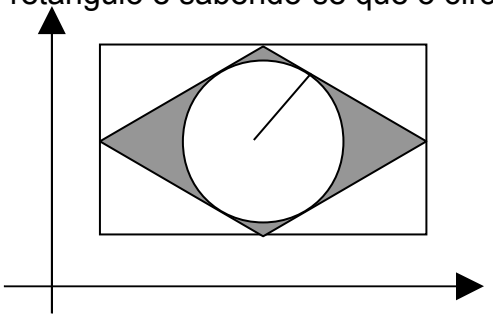
d) not (3 == 3) || not (4 /= 5) || 7 < 15 && 7 > 2

2. Dado um ponto P(x,y) do plano cartesiano, defina funções que descrevam a sua pertinência nas regiões cinzas das figuras abaixo:

i) A região R1 (região cinza), do retângulo dado pelas coordenadas do canto superior esquerdo e do canto inferior direito, como mostrado na figura abaixo:



ii) A região R2 do losango (região cinza), sendo dados os pontos E e D do retângulo e sabendo-se que o círculo é tangente aos lados do losango.

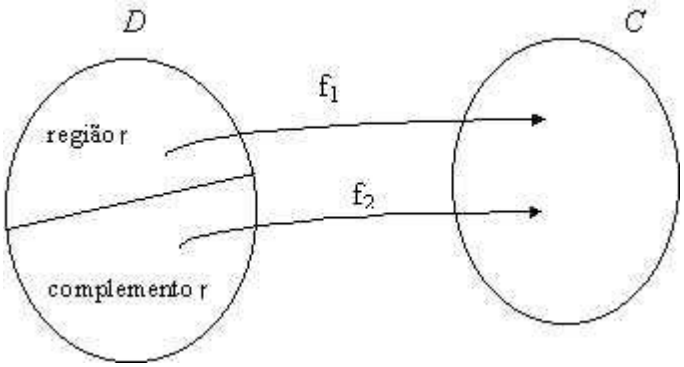


7. DEFINIÇÕES CONDICIONAIS

7.1. INTRODUÇÃO

Sabemos de nosso conhecimento matemático que algumas funções não são contínuas em um domínio e que, portanto, possuem várias definições.

Em muitos casos, o domínio D de uma função está dividido em regiões disjuntas que se complementam e, para cada uma dessas regiões, existe uma expressão que define o seu mapeamento no contra-domínio. Podemos representar esta situação pela figura abaixo:



Exemplo 1 - Considere a função que determina o valor da passagem aérea de um adulto, para um determinado trecho, por exemplo, Vitória-Manaus, considerando a sua idade. Pessoas com idade a partir de 60 anos possuem um desconto de 40% do valor. Considere ainda que a passagem para o trecho considerado custe R$ 600,00.

Temos aqui duas formas de calcular o valor da passagem de uma pessoa, dividindo o domínio em dois subconjuntos. O subconjunto dos adultos com menos de 60 anos e o subconjunto dos demais.

Podemos definir as duas funções a seguir:

```
vpass1 = 600

vpass2 = vpass1 * 0.6
```

Para usar uma das definições, temos que explicitamente escolher a que se aplica ao nosso caso.

Exemplo 2 - Considere a função que associa com um determinado rendimento o Imposto de Renda a ser pago. Até um determinado valor, o contribuinte não paga imposto, e a partir de então o rendimento é dividido em faixas (intervalos), aos quais se aplicam diferentes taxas. Suponha a tabela hipotética abaixo.

Faixa	alíquota	Desconto
inferior ou igual a 10.800	0	0
entre 10.801 e 20.000	10	1000
entre 20.001 e 30.000	20	1500
acima de 30.000	25	1800

Para descrever as várias definições e os correspondentes subdomínios, poderíamos escrever separadamente cada definição, construindo, portanto várias funções, e deixar que o usuário escolha qual usar. Claro que isto traria muitos inconvenientes pois estaríamos deixando uma escolha mecânica na mão do usuário, que além de sobrecarregá-lo com tarefas desnecessárias, ainda estaria expondo-o ao erro por desatenção. Mas vamos lá construir as funções independentes.

```
ir1 s = 0

ir2 s = s * 0.1 - 1000

ir3 s = s * 0.2 - 1500

ir4 s = s * 0.25 - 1800
```

Agora, para usá-las, o usuário pega o seu salário, olha a tabela e seleciona qual função aplicar.

A escolha de qual definição usar para uma dada situação é em si, um tipo de computação. Podemos descrever essa computação com expressões condicionais, vamos deixar que o computador escolha. Descrevemos cada subdomínio com a respectiva função aplicável e deixemos que ele escolha a definição a aplicar, dependendo do valor fornecido. Vejamos então como isso pode ser feito nas seções subseqüentes.

7.2. A ESTRUTURA IF-THEN-ELSE

Uma expressão condicional construída com ***if-then-else*** possui a seguinte sintaxe:

if <expressão lógica> **then** <expressão 1> **else** <expressão 2>

onde:

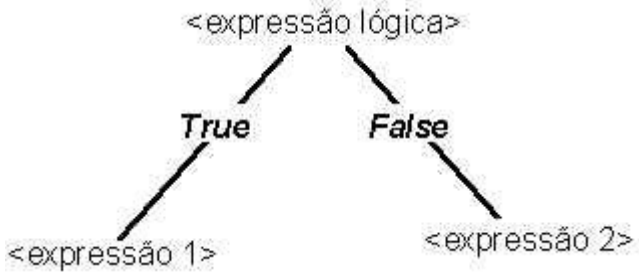
<expressão lógica>	Uma expressão descrevendo uma condição a ser satisfeita, envolvendo operadores relacionais e operadores lógicos.

<expressão1> e <expressão2>	1. Expressões descrevendo um valor a ser produzido como resposta à entrada fornecida e, como a expressão total tem que ser de um único tipo, as duas expressões devem ser do mesmo tipo. 2. Cada uma destas expressões pode ser inclusive outra condicional, dentro da qual pode haver outras e assim sucessivamente. 3. Quando a <expressão lógica> é avaliada para ***True*** o valor resultante será o que for obtido pela avaliação da <expressão 1> caso contrário será o obtido pela avaliação da <expressão 2>

Para a função que calcula o valor da passagem aérea podemos então construir a seguinte definição:

```
vpass x = if x < 60 then 600 else 360
```

Árvore de decisão: Podemos representar as expressões condicionais através de uma notação gráfica denominada de árvore de decisão. É importante considerar que este tipo de representação é uma ferramenta importantíssima para estruturarmos a solução de problemas que requerem expressões condicionais.



Exemplo 3 – Definir a função que determina o valor absoluto de um número. Sabemos que esta função se define em dois subdomínios:

subdomínio	expressão
x < 0	- x
x >= 0	x

Como só temos duas possibilidades, podemos codificar da seguinte maneira:

absoluto x	=	**if** x < 0 **then** -x **else** x

Para concluir esta apresentação voltemos ao nosso exemplo 2 que define a função para cálculo do Imposto de Renda. O domínio neste caso deve ser quebrado em quatro subdomínios e para cada um deles construiremos uma expressão.

domínio	função
s ≤ 10800	ir1 s
pert s 10801 20000	ir2 s
pert s 20001 30000	ir3 s
s > 30000	ir4 s

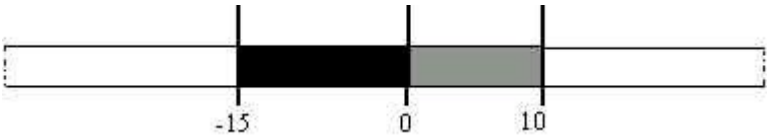
Para a dodificação, precisaremos quebrar sucessivamente o domínio da função em intervalos. A determinação dos intervalos é realizada da seguinte maneira: primeiro dividimos o domínio entre o primeiro intervalo e o restante, que por sua vez, é dividido entre o segundo intervalo e o seu restante e assim sucessivamente.

A codificação final pode ser:

```
ir s = if s <= 10800
       then ir1
       else if  pert s 10800 20000
            then ir2
            else if pert s 20001 30000
                 then ir3
                 else ir4
       where
          ir1 = 0
          ir2 = s * 0.1 - 1000
          ir3 = s * 0.2 - 1500
          ir4 = s * 0.25 - 1800
          pert x a b = x>=a && x<=b
```

7.2.1 USANDO O IF-THEN-ELSE

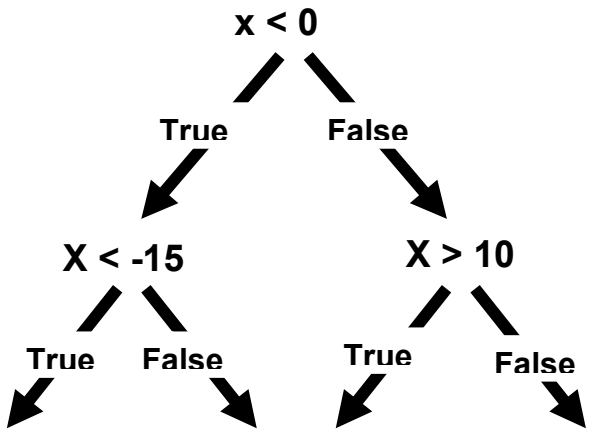
EXEMPLO 1: Considere um mapeamento de valores numéricos onde o domínio se divide em 4 regiões, cada uma das quais possui diferentes formas de mapeamento. As regiões são apresentadas na figura abaixo, numeradas da esquerda para direita. Observe ainda que as extremidades são abertas. Considere ainda a seguinte tabela de associações:



região	mapeamento desejado
região 1	o dobro de x

região 2	o sucessor de x
região 3	o quadrado de x
região 4	o simétrico do quadrado de x

Podemos analisar as regiões através do seguinte diagrama:

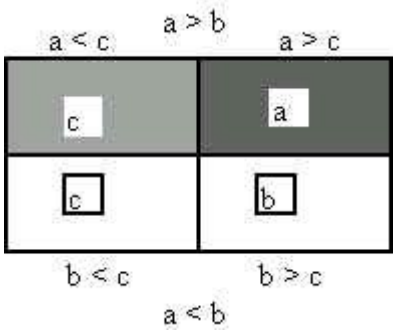


O que nos levará à seguinte definição:

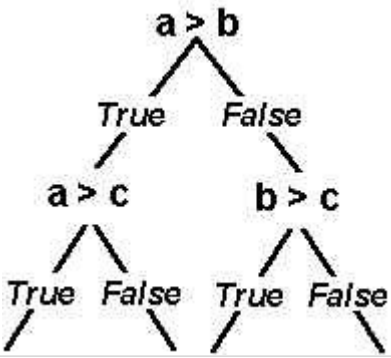
```
f x = if x < 0
      then if x < (-15)
           then 2 * x
           else x + 1
      else if x < 10
           then x ^2
           else - (x ^ 2)
```

Exemplo 2**:** Dados três números inteiros distintos, determinar o maior deles.

Podemos explorar uma solução da seguinte maneira. Considere um retângulo e divida-o horizontalmente em 2 partes, a parte de cima representa as situações onde a > b e a de baixo aquelas onde b > a. Divida agora o retângulo verticalmente, em cada uma das regiões anteriores surgirão 2 metades. Na de cima, representamos agora a relação entre a e c. Na de baixo, a relação entre b e c.



Explorando as relações entre os números



Representando as relações através de uma árvore de decisão

Traduzindo a árvore de decisão para Hugs, chegamos à seguinte definição:

```
maior a b c =  if  a > b
               then if a > c
                    then  a
                     else   c
               else if  b > c
                    then   b
                    else   c
```

7.3 DEFINIÇÕES PROTEGIDAS (guarded commands): A estrutura **IF-THEN-ELSE** foi apresentada por primeiro por questões históricas, por tratar-se de uma forma pioneira de escrever definições condicionais. Entretanto, algumas vezes podemos lançar mão de estruturas mais simples e mais legíveis.

As definições protegidas, também conhecidas por "*guarded commands*" permitem que se escreva para uma mesma função, várias definições, cada uma delas protegida por uma expressão lógica.

```
<nome da função> <parâmetros> | <proteção 1> = <definição 1>
                              | <proteção 2> = <definição 2>
                              | <proteção 3> = <definição 3>
                              . . .
                              | <proteção n> = <definição n>
                             "[" | otherwise    = <definição n + 1> "]"
```

A última clausula da definição é opcional, por este motivo está apresentada dentro de colchetes.

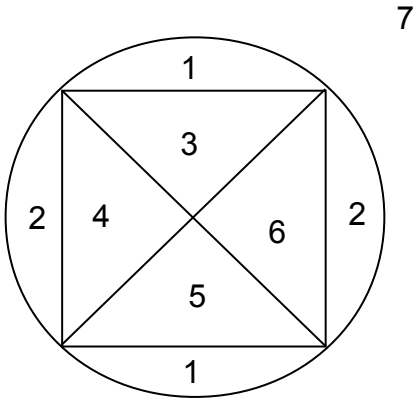
Vejamos como podemos reescrever a definição da nossa função "ir" para cálculo do imposto de renda.

```
ir' s | s<=10800           = ir1
      | pert s 10800 20000 = ir2
      | pert s 20001 30000 = ir3
      |otherwise           = ir4
      where
         ir1 = 0
         ir2 = s * 0.1 - 1000
         ir3 = s * 0.2 - 1500
         ir4 = s * 0.25 - 1800
```

Exercícios

1. Reescreva, usando definições protegidas. A definição da função que determina o maior de 3 números inteiros fornecidos.

2. Sejam C(x_1,y_1) o centro da circunferência de raio *r* e também do quadrado de lados paralelos aos eixos cartesianos e inscrito na circunferência. Escreva uma função que descreva a **região de pertinência do ponto P** na figura abaixo, dados C(x_1,y_1) , *r* e o ponto P (x,y),



8. O TESTE DE PROGRAMAS

8.1. INTRODUÇÃO: Não basta desenvolver um programa para resolver um dado problema. É preciso garantir que a solução esteja correta. Muitos erros podem ocorrer durante o desenvolvimento de um programa e, portanto temos que garantir que o programa que irá ser executado está livre de todos eles. Ao conceber a solução podemos nos equivocar e escolher caminhos errados. Precisamos eliminar esses equívocos. Ao codificarmos a nossa solução podemos cometer outros erros ao não traduzirmos corretamente nossa intenção. Esses erros podem ocorrer por um mau entendimento dos elementos da linguagem ou até mesmo por descuido, o certo é que eles ocorrem. Uma estratégia muito útil, mas não infalível, é o teste de programa.

Em sua essência, o teste de programa consiste em submeter um programa ao exercício de algumas instâncias do problema e comparar os resultados esperados com os resultados obtidos.

8.2. O PROCESSO DE TESTE: Em primeiro lugar devemos escolher as instâncias apropriadas, não basta escolhê-las aleatoriamente. A seguir devemos determinar, sem o uso do programa, qual o valor que deveria resultar quando o programa for alimentado com essas instâncias. O passo seguinte consiste em submeter cada instância ao programa e anotar o resultado produzido por ele. Finalmente devemos comparar cada valor esperado com o valor produzido e descrever qual o tipo de ocorrência.

Um exemplo: Considere o problema de identificar se um dado ponto está ou não localizado no primeiro quadrante do espaço cartesiano. Considere ainda a seguinte definição:

primquad x y	=	(x >= 0) && (y >= 0)

Precisamos agora verificar se ela atende nossa intenção. Para tanto devemos escolher algumas instâncias, prever o resultado esperado e em seguida submeter ao HUGS, para ver o que acontece.

Que pares de valores deveremos escolher? Bom, vamos escolher uns pares usando a seguinte estratégia: um par onde x é maior que y, outro onde y seja maior que x e um terceiro em que os dois sejam iguais. Gerando uma planilha como apresentada a seguir.

x	y	resultado esperado	resultado obtido	diagnóstico
-5	-2	False		
-2	-5	False		
5	5	True		

Podemos agora submeter as instâncias à avaliação do sistema, obtendo a seguinte interação:

```
? primquad (-5) (-2)
False
? primquad (-2) (-5)
False
? primquad 5 5
True
?
```

Podemos agora completar o preenchimento de nossa planilha, obtendo a tabela a seguir:

x	y	resultado esperado	resultado obtido	diagnóstico
-5	-2	False	False	sucesso
-2	-5	False	False	sucesso
5	5	True	True	sucesso

Verificando as diversas linhas da coluna “Diagnóstico”, parece que nosso programa está correto. Veja que ele passou com sucesso em todos os testes!

Que pena que não seja verdade. Apesar de passar em todos os testes a que foi submetido, ele não funciona corretamente. Tudo que podemos afirmar neste instante é que para os valores usados, o programa funciona corretamente. E para os outros valores, será que funciona corretamente?

Outros valores? Quais?

8.3. PLANO DE TESTE: Para escolher os valores que usaremos, antes de mais nada devemos identificar as classes de valores que serão relevantes para o teste, em um segundo instante podemos então escolher os representantes destas classes. Quando temos um parâmetro, os possíveis valores a serem usados são todas as constantes do domínio. Para o caso de um parâmetro do tipo ***int,*** existem 65536 valores diferentes. Testar nosso programa para todos esses valores implicaria em determinar a mesma quantidade de resultados esperados e em seguida digitar e submeter esta mesma quantidade de instâncias do problema para o sistema e ainda depois conferir um a um os resultado obtidos com os esperados. Enfim, um trabalho imenso. Imagine agora se fossem dois parâmetros? A quantidade de pares seria da ordem de 4.29497 x 10^09 (algo da ordem de quatro bilhões). E para 3 parâmetros? E para ***n*** parâmetros? Números cada vez mais enormes. Por este caminho, testar programas seria inviável.

Felizmente não precisamos de todos eles, basta identificar as classes distintas que importam para o problema, ou seja, as classes de equivalência relevantes. Isto pode ser obtido analisando o enunciado estendido do problema.

No caso de nosso exemplo anterior, analisando melhor a definição, podemos identificar que por definição, o espaço cartesiano se divide em quatro regiões. A primeira, onde ambos as coordenadas são positivas, denominamos de primeiro quadrante. A segunda, onde a coordenada x é negativa e a y positiva, denominamos de segundo quadrante. O terceiro quadrante corresponde ao espaço onde ambas as coordenadas são

negativas. Ainda temos o quarto quadrante onde a coordenada x é positiva e a y é negativa. E os pontos que ficam em um dos eixos ou na interseção destes, qual a classificação que eles têm? Bom, podemos convencionar que não estão em nenhum dos 4 quadrantes descritos. Resumindo, nosso plano de teste deve contemplar estas situações, conforme é ilustrado na tabela a seguir.

x	**y**	**quadrante**
positivo	positivo	primeiro
negativo	positivo	segundo
negativo	negativo	terceiro
negativo	positivo	quarto
nulo	qualquer não nulo	eixo das ordenadas
qualquer não nulo	nulo	eixo das abscessos
nulo	nulo	origem

É bom observar ainda que este plano pode e deve ser preparado antes mesmo de elaborar o programa, para fazê-lo, precisamos apenas da especificação detalhada. Além disso, este plano não precisa ser feito pelo programador responsável pela elaboração da solução, qualquer outro programador, de posse do enunciado detalhado, pode se encarregar da tarefa. Este tipo de plano serve para alimentar o teste denominado de ***caixa preta***. Esta denominação se deve ao fato de não ser necessário conhecermos a estrutura da solução para elaborar o plano. Outro aspecto positivo da elaboração do plano o mais cedo possível é que contribuí para um melhor entendimento do problema.

Voltando ao nosso exemplo, podemos agora elaborar a nossa planilha de teste considerando as classes de equivalência a serem definidas. Uma questão que surge é como escolhemos o representante de uma classe? Existem melhores e piores? No nosso caso, como pode ser qualquer valor positivo ou negativo, podemos escolher valores de um dígito apenas, no mínimo escreveremos e digitaremos menos.

x	**y**	**resultado esperado**	**resultado obtido**	**diagnóstico**
2	3	True		
-2	3	False		
-2	-3	False		
2	-3	False		
0	3	False		
0	-3	False		
2	0	False		
-2	0	False		
0	0	False		

8.4. REALIZANDO O TESTE: Vejamos agora o resultado de nossa interação com o HUGS.

```
? primquad 2 3
True
? primquad (-2) 3
False
? primquad (-2) (-3)
False
? primquad 2 (-3)
False
primquad 0 (-3)
False
? primquad 0 3
True
? primquad 0 (-3)
False
? primquad 2 0
True
? primquad (-2) 0
False
? primquad 0 0
True
?
```

Voltemos agora para nossa planilha e vamos preenchê-la na coluna de resultado obtido e diagnóstico.

x	y	resultado esperado	resultado obtido	diagnóstico
2	3	True	True	sucesso
-2	3	False	False	sucesso
-2	-3	False	False	sucesso
2	-3	False	False	sucesso
0	3	False	True	falha
0	-3	False	False	sucesso
2	0	False	True	falha
-2	0	False	False	sucesso
0	0	False	True	falha

8.5. Depuração**:** Uma vez testado o programa e identificado que ocorreram instâncias para as quais a nossa definição não está fazendo a associação correta, ou seja, o valor obtido é diferente do esperado, devemos passar a uma nova fase. Depurar um programa é um processo que consiste em buscar uma explicação para os motivos da falha e posteriormente corrigi-la. Obviamente isto também não é um processo determinante e nem possuímos uma fórmula mágica. Ao longo de nossa formação de programadores iremos

aos poucos incorporando heurísticas que facilitem esta tarefa. Por enquanto é muito cedo para falarmos mais do assunto e vamos concluir com um fechamento do problema anterior. Após concluir as modificações devemos testar o programa novamente.

Depurando nossa solução - Podemos concluir por simples inspeção da nossa última planilha (aquela com todas as colunas preenchidas) que nossa solução está incorreta. Uma interpretação dos resultados nos leva à hipótese de que a nossa solução considera que quando o ponto se localiza na origem ou em um dos eixos positivos, a nossa definição está considerando que eles estão no primeiro quadrante.

Passo seguinte, verificar se de fato nossa definição incorre neste erro. Em caso afirmativo, corrigi-la e a seguir, resubmetê-la aos testes.

Observando a nossa definição inicial, podemos concluir que de fato nossa hipótese se confirma.

primquad x y	=	(x >= 0) && (y >= 0)

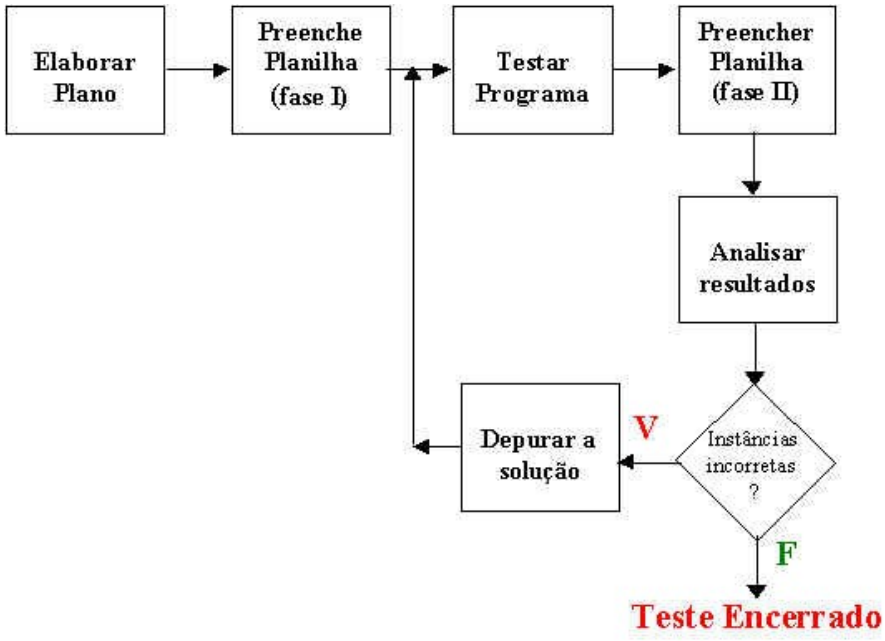
Podemos então modificá-la para obter uma nova definição, que esperamos que esteja correta.

primquad x y	=	(x > 0) && (y > 0)

Agora é submetê-la novamente ao teste e ver o que acontece!

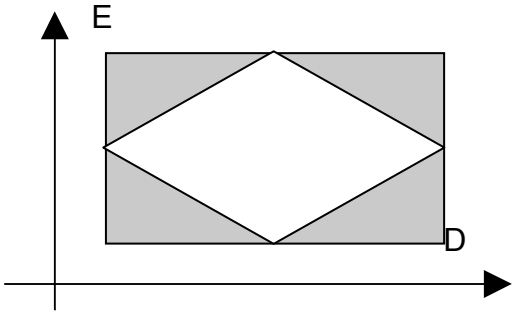
8.6. UMA SÍNTESE DO PROCESSO: O processo é, como vimos, repetitivo e só se encerra quando não identificarmos mais erros. O diagrama ilustra o processo como um todo.

Mas lembre-se, isto ainda não garante que seu programa esteja 100% correto! Quando não identificamos erro, apenas podemos concluir que para as instâncias que usamos o nosso programa apresenta os resultados esperados.



Exercícios:

1. Dado um ponto P(x,y) do plano cartesiano, defina funções que descrevam a sua pertinência na região cinza da figura abaixo (a região R1 - região cinza - do retângulo dado pelas coordenadas do canto superior esquerdo e do canto inferior direito. Faça o plano de teste para o seu programa:



9. RESOLVENDO PROBLEMAS - OS MOVIMENTOS DO CAVALO

9.1. INTRODUÇÃO: Considere o jogo de xadrez, onde peças são movimentadas em um tabuleiro dividido em 8 linhas e oito colunas. Considere ainda os movimentos do cavalo, a partir de uma dada posição, conforme diagrama a seguir, onde cada possível movimento é designado por ***mi*.** No esquema, o cavalo localizado na posição (5, 4) pode fazer oito movimentos, onde o primeiro deles, ***m1***, levaria o cavalo para a posição (7,5).

8								
7								
6				m3		m2		
5			m4				m1	
4					**C**			
3			m5				m8	
2				m6		m7		
1								
	1	2	3	4	5	6	7	8

9.2. PROBLEMA 1: Escreva uma função que determina se, a partir de uma dada posição, o cavalo pode ou não realizar o primeiro movimento. Vamos chamá-la de ***pmov***, e denominar seus parâmetros (a posição corrente), de ***x*** e ***y***. Eis alguns exemplos de uso de nossa função e os valores esperados:

instância	**resultado**
pmov 5 4	True
pmov 8 1	False
pmov 1 1	True
pmov 1 8	False

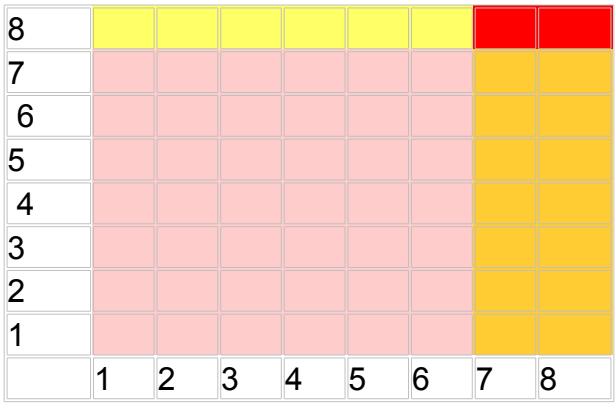
9.2.1. Solução - A solução pode ser encontrada através da construção de uma expressão booleana que avalie se a nova posição, ou seja, aquela em que o cavalo seria posicionado pelo primeiro movimento, está dentro do tabuleiro. Como o cavalo, no primeiro movimento, anda duas casas para direita e uma para cima, basta verificar se as coordenadas da nova posição não ultrapassam a oitava fileira (linha ou coluna).

Codificando em HUGS, temos então:

```
pmov x y  =  (x + 2 <= 8 ) && (y + 1 <= 8)
```

9.2.2. TESTANDO A SOLUÇÃO - Como já discutimos anteriormente, para verificar a correção de nossa solução, devemos submetê-la a um teste. Para tanto devemos escolher as classes de equivalências relevantes e, a partir delas, produzir a nossa planilha de teste. Olhando para a especificação do problema, podemos identificar 4 conjuntos de

valores que se equivalem para os fins do nosso programa, como podemos observar na figura abaixo:



9.2.3. Estendendo o Problema - Podemos fazer o mesmo para todos os demais movimentos, obtendo com isso as seguintes definições:

pmov x y	=	(x + 2 <= 8) && (y + 1 <= 8)
smov x y	=	(x + 1 <= 8) && (y + 2 <= 8)
tmov x y	=	(x - 1 >= 1) && (y + 2 <= 8)
qmov x y	=	(x - 2 >= 1) && (y + 1 <= 8)
qtmov x y	=	(x - 2 >=1) && (y - 1 >= 1)
sxmov x y	=	(x - 1 >= 1) && (y - 2 >= 1)
stmov x y	=	(x + 1 <= 8) && (y - 2 >= 1)
omov x y	=	(x + 2 <= 8) && (y - 1 >= 1)

9.2.4. Identificando Abstrações - Podemos agora indagar, olhando para as oito definições, sobre a ocorrência de algum conceito geral que permeie todas elas. Poderíamos também ter feito isso antes. Como não o fizemos, façamo-lo agora. Podemos observar que todas elas avaliam duas expressões e que ambas testam fronteiras que podem ser margem direita, margem esquerda, margem superior ou margem inferior. Podemos observar ainda, que o par margem direita e margem superior testam o mesmo valor 8, assim como ocorre com as duas outras, que testam o valor 1. Com isso podemos definir duas novas funções, ***f*** e ***g***, para testar estes limites. Agora, as nossas definições anteriores podem ser reescritas, usando as duas abstrações identificadas.

pmov x y	=	f (x + 2) && f(y + 1)
smov x y	=	f (x + 1) && f (y + 2)
tmov x y	=	g (x - 1) && f (y + 2)
qmov x y	=	g (x - 2) && f (y + 1)
qtmov x y	=	g (x - 2) && g (y – 1)
sxmov x y	=	g (x - 1) && g (y – 2)
stmov x y	=	f (x + 1) && g (y – 2)
omov x y	=	f (x + 2) && g (y – 1)
f w	=	w <= 8
g w	=	w >= 1

9.2.5. Análise da Solução - O que será que ganhamos com esta nova forma de descrever a nossa solução? Podemos indicar pelo menos três indícios de vantagem na nova solução:

1. Clareza - Na medida em que agora está explicitado, que todas as oito funções para verificar os movimentos possuem estrutura semelhante e que todas estão usando funções para verificar a ultrapassagem das bordas;
2. Manutenção - Se nosso tabuleiro mudasse, ou seja, passasse a ter 9 linhas por nove colunas, bastaria alterar a função ***f*** e tudo estaria modificado, ao invés de termos que alterar as oito definições.
3. Reuso - As duas funções que testam as bordas poderiam ser usadas para construir funções para avaliar o movimento de outras peças do jogo de xadrez.

9.3. PROBLEMA 2: Sabemos que para cada posição alguns movimentos podem ser realizados e outros não. Como ordenamos os movimentos no sentido anti-horário, gostaríamos de obter, para uma dada posição, dos movimentos que podem ser realizados, aquele que possui o menor índice. Vejamos o esquema abaixo.

8	m4				m1			C1
7			C3			m5		
6	m5				m8		m6	
5		m6		m7				
4								
3		m2					m3	
2			m1			m4		
1	C4							C2
	1	2	3	4	5	6	7	8

Podemos observar que o cavalo C1 só pode fazer os movimentos m5 e m6 e que o de menor índice é m5. Já o cavalo C2 só pode fazer os movimentos m3 e m4 e que o de menor índice é o m3. Enquanto isso o cavalo C3 pode fazer os movimentos m1, m4, m5, m6, m7 e m8. Para este caso o movimento de menor índice é o m1.

Vamos chamar esta função de ***qualmov*** e, como no problema anterior, os parâmetros serão as coordenadas da posição atual do cavalo. Eis alguns exemplos de uso de nossa função:

Instância	resultado
qualmov 8 1	3
qualmov 8 8	5
qualmov 3 7	1
qualmov 1 1	1

9.3.1. SOLUÇÃO - Bem, como já sabemos, para verificar se um dado movimento ***mi*** é possível, basta arranjar um meio de sair verificando um-a-um os movimentos, a partir do primeiro (m1) e encontrar o primeiro que pode ser realizado. Quando isso ocorrer podemos fornecer como resposta o seu índice. Podemos construir para isso uma árvore de decisão. Na raiz da árvore estará a pergunta

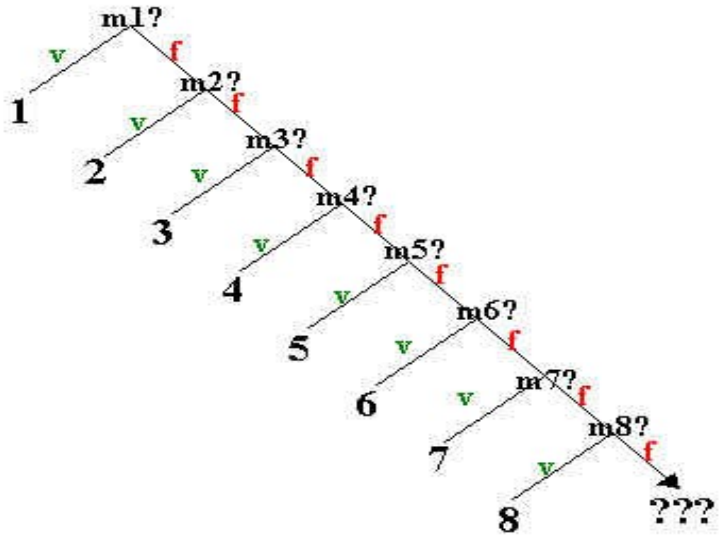
"é possível realizar o movimento m1"?

Em caso afirmativo (braço esquerdo da árvore), mapeamos no valor 1 e em caso negativo (braço direito), o que devemos fazer? Bom, aí podemos começar uma nova árvore (na verdade uma sub-árvore), cuja raiz será:

"é possível realizar o movimento m2"?

E daí, prosseguimos até que todos os movimentos tenham sido considerados.

A árvore resultante será:



9.3.2. CODIFICANDO A SOLUÇÃO - Vamos então explorar os recursos da linguagem para transformar nosso plano em um programa que de fato possa ser "entendido" pelo nosso sistema de programação (HUGS). Como podemos observar temos aqui o caso de uma função que não é contínua para o domínio do problema. Pelo que sabemos até então, não dá para expressar a solução como uma única expressão simples. Resta-nos o recurso das expressões condicionais. Para verificar se um dado movimento é satisfeito podemos usar as funções que construímos anteriormente e com isso obtemos a seguinte definição:

```
qualmov x y = if pmov x y then 1
              else if smov x y then 2
              else if tmov x y then 3
              else if qmov x y then 4
              else if qtmov x y then 5
              else if sxmov x y then 6
              else if stmov x y then 7
              else if omov x y then 8
              else 0
```

9.3.3. Análise da Solução - EM PRIMEIRO LUGAR, INCLUÍMOS A RESPOSTA IGUAL A ZERO (0) QUANDO O MOVIMENTO M8, O ÚLTIMO A SER AVALIADO, RESULTA EM FRACASSO. PARA QUE SERVE ISSO? ACONTECE QUE SE A POSIÇÃO DE ENTRADA NÃO FOR VÁLIDA, OU SEJA, UMA OU AMBAS AS COORDENADAS NÃO PERTENCEREM AO INTERVALO [1, 8], NENHUM MOVIMENTO SERIA VÁLIDO E SE NÃO PROVIDENCIARMOS UMA RESPOSTA ALTERNATIVA, NOSSA FUNÇÃO SERIA PARCIAL. MAS ISTO RESOLVE DE FATO NOSSO PROBLEMA? O QUE OCORRERIA SE A POSIÇÃO DE ENTRADA FOSSE (0, 0)? BOM, NOSSA FUNÇÃO DETERMINARIA QUE O PRIMEIRO MOVIMENTO PODERIA SER REALIZADO E ISTO NÃO É VERDADE. A INVENÇÃO DE UM RESULTADO EXTRA PARA INDICAR QUE NÃO HÁ SOLUÇÃO POSSÍVEL, TRANSFORMANDO UMA FUNÇÃO PARCIAL EM UMA FUNÇÃO TOTAL, PARECE SER BOA, MAS COMO FOI FEITA NÃO RESOLVEU. EM GERAL O MELHOR NESTAS SITUAÇÕES É PRECEDER TODA E QUALQUER TENTATIVA DE DETERMINAR A SOLUÇÃO ADEQUADA, POR UMA AVALIAÇÃO DA VALIDADE DOS DADOS DE ENTRADA. NESTE CASO, BASTARIA VERIFICAR SE OS DOIS ESTÃO NO INTERVALO [1, 8]. VAMOS CONSTRUIR AQUI UMA FUNÇÃO QUE AVALIA A PERTINÊNCIA DE UM VALOR A UM INTERVALO NUMÉRICO, CONFORME DEFINIÇÃO A SEGUIR:

```
pert x a b = (x >= a) && (x<=b)
```

Especulando um pouco mais sobre a nossa solução, podemos observar que o movimento ***m8***, jamais ocorrerá! Analisando os possíveis movimentos chegaremos à conclusão de que para nenhuma posição, o oitavo é o único movimento possível. Sugerimos fortemente que o leitor prove este teorema. Portanto a solução final pode ser:

```
qualmov x y = if not (pert x 1 8)
              || not (pert y 1 8) then 0
              else if pmov x y then 1
              else if smov x y then 2
              else if tmov x y then 3
              else if qmov x y then 4
              else if qtmov x y then 5
              else if sxmov x y then 6
              else 7
```

9.4. REVISITANDO O PROBLEMA 1: Observando a solução encontrada para o problema 1, constatamos que embora a noção de movimento do cavalo seja única, quem precisar saber se um dado movimento é válido, precisará conhecer o nome das oito funções. Embora seja cedo para falarmos de interface homem-máquina, já dá para dizer que estamos sobrecarregando nosso usuário ao darmos oito nomes para coisas tão parecidas. Será que temos como construir uma só função para tratar o problema? Vamos reproduzir aqui a interface das oito:

pmov x y
smov x y
tmov x y
qmov x y
qtmov x y
sxmov x y
stmov x y
omov x y

Propositadamente escrevemos o nome delas com um pedaço em vermelho e outro em preto. Seria alguma homenagem à algum time que tem essas cores? Na verdade estamos interessados em destacar que a pequena diferença nos nomes sugere que temos uma mesma função e que existe um parâmetro oculto. Que tal explicitá-lo? Podemos agora ter uma função com 3 parâmetros, sendo o primeiro deles para indicar o número do movimento que nos interessa. A interface agora seria:

mov m x y

Agora, por exemplo, para solicitar a avaliação do sétimo movimento para um cavalo em (3, 4), escrevemos:

? mov 7 3 4
True
?

Muito bem, e como codificaremos isso?

9.4.1. SOLUÇÃO - Precisamos encampar em nossa solução o fato de que a nossa função possui diferentes formas de avaliação, para diferentes valores do domínio, algo parecido com a solução do problema 2, ou seja, a nossa função não é continua e portanto temos que selecionar qual a definição apropriada para um determinado valor de m. Devemos construir uma árvore de decisão. Aqui deixamos esta tarefa a cargo do leitor e passamos direto à codificação conforme apresentamos a seguir:

```
mov m x y  =  if not (pert m 1 8) ||
                 not (pert x 1 8) ||
                 not (pert y 1 8) then
              False
              else if m == 1 then pmov
              else if m == 2 then smov
              else if m == 3 then tmov
              else if m == 4 then qmov
              else if m == 5 then qtmov
              else if m == 6 then sxmov
              else if m == 7 then stmov
              else omov
              where
                     pmov = ...
                     smov = ...
                     tmov = ...
                     ...
```

9.4.2. Análise da solução - Ao contrário da solução do problema 2, onde necessariamente a validade dos movimentos tinha que ser verificada do primeiro para o último, pois nos interessava saber qual o primeiro possível de ser realizado, o leitor pode observar que esta ordem aqui não é necessária. Tanto faz se perguntamos primeiro se m=7 ou se m=3. Será que podemos tirar algum proveito disso? Alertamos o iniciante, que devemos sempre identificar propriedades internas do problema e explorá-las adequadamente. Qual a influência desta ordem na eficiência da avaliação de uma expressão submetida ao HUGS? Para responder, basta lembrar que as condições são avaliadas seqüencialmente. Por exemplo, se m=8, teremos que avaliar 8 condições, se m=1 faremos 2 avaliações e se m está fora do domínio faremos uma avaliação. Ou seja, no pior caso faremos 8 avaliações e no melhor caso uma (1). Em média, ao longo do uso da função, assumindo uma distribuição uniforme dos valores de m, faremos 4 avaliações. E se o intervalo fosse de 100 elementos distintos, quantas avaliações de condições faríamos em média? Será possível elaborar soluções onde este número de avaliações seja menor?

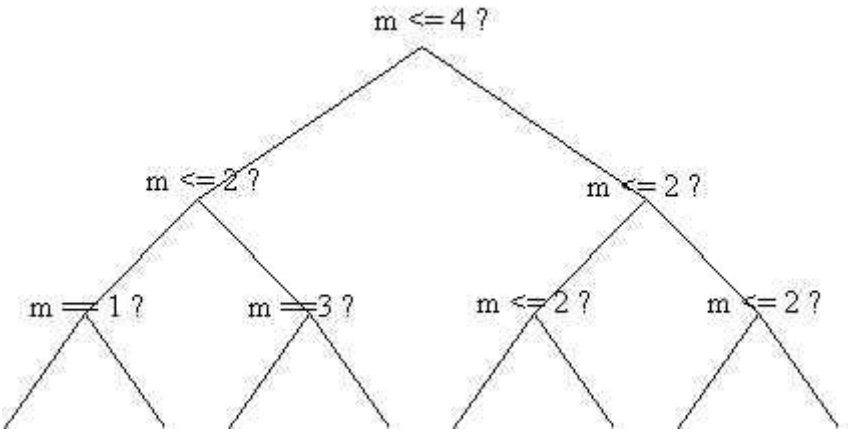
9.4.3. A resposta é sim! Já que a ordem das avaliações não importa, podemos buscar uma ordem mais conveniente para reduzir o número de avaliações por cada instância avaliada.

A idéia geral para estes casos é obtida a partir de um conceito de vasta utilidade na Computação. Falamos de ***árvore binária*** e o leitor por certo ouvirá falar muito dela ao longo da vida enquanto profissional da área.

O caminho consiste em dividir o intervalo considerado ao meio e fazer a primeira pergunta, por exemplo, m<=4? Dividindo em 2 o intervalo de comparações, para cada um destes podemos novamente dividir ao meio, até que não mais tenhamos o que dividir, como ilustramos no diagrama a seguir, onde cada linha representa um passo da divisão dos intervalos.

<1	1	2	3	4	5	6	7	8	>8
<1	1	2	3	4	5	6	7	8	>8

Que também podemos representar por:



A codificação ficará então:

```
mov m x y = if not(pert m 1 8)||not(pert x 1 8)||not (pert y 1 8)
            then False
            else if m <= 4
                 then if m<= 2
                      then if m== 1
                           then pmov
                           else smov
                      else if m==3
                           then tmov
                           else qmov
                 else if m<= 6
                      then if m==5
                           then qtmov
                           else sxmov
                      else if m == 7
                           then stmov
                           else omov
            where
               pmov = ...
               smov = ...
               tmov = ...
               ...
```

Se fizermos uma análise das possibilidades veremos que para qualquer valor de m o número máximo de avaliações que faremos será sempre igual a quatro! E se fosse 100 valores para m, qual seria o número máximo de comparações? E para 1000? E 1000000?

O número de comparações, seguindo este esquema, é aproximadamente igual ao logaritmo do número de valores na base 2. Portanto para 100 teríamos 7, para 1000 teríamos 10 e para um milhão teríamos 20. Veja que é bem inferior ao que obtemos com o esquema anterior, também denominado de linear. Confira a comparação na tabela abaixo.

número de valores	**esquema linear** (número médio)	**esquema binário** (número máximo)
8	4	4
100	50	7
1000	500	10
1000000	500000	20

Exercícios:

1. Escreva uma função que determina se, a partir de uma dada posição e de um movimento válido (1 a 8), o cavalo pode ou não realizar o movimento desejado. Se puder, sua função deve responder em qual linha do tabuleiro o novo ponto, obtido após a realização do movimento, estará situado. Sua função deve se chamar movL e uma instância dela seria movL 1 4 5 ⇒ 5.

2. Faça uma função similar à movL, mas agora sua função deve se chamar movC e deve responder a coluna correspondente da nova posição do cavalo no tabuleiro, após a realização do movimento desejado.

10. TUPLAS

10.1. INTRODUÇÃO

Até então trabalhamos com valores elementares. Ou seja, valores atômicos, indivisíveis no que diz respeito as operações de seu tipo de dados. Por exemplo, ***True***, ***523***, ***8.234***, são valores elementares dos tipos Boolean, Integer e Float, respectivamente. Não podemos, por exemplo, nos referir, dentro da linguagem, ao primeiro algarismo do valor ***523***. Dizemos que esses são tipos primitivos da linguagem. Além desses três, a linguagem Haskell provê ainda o tipo ***char***. O elenco de tipos primitivos de uma linguagem pode ser entendido como o alicerce da linguagem para a descrição de valores mais complexos, que são denominados genericamente de ***valores estruturados***.

Os problemas que pretendemos resolver estão em universos mais ricos em abstrações do que esse das linguagens. Para descrever esses mundos, precisamos usar abstrações apropriadas para simplificar nosso discurso. Por exemplo, quando quero saber onde alguém mora, eu pergunto: Qual o seu ***endereço***? Quando quero saber quando um amigo nasceu, eu pergunto: Qual a ***data*** do seu nascimento? E quando quero saber para onde alguém vai deslocar o cavalo no jogo de xadrez, eu pergunto: Qual a nova ***posição***? Os nomes *endereço*, *data* e *posição* designam valores estruturados. Uma data tem três partes: dia, mês e ano. Um endereço pode ter quatro: rua, número, complemento e bairro. Já a posição no tabuleiro tem duas partes, linha e coluna.

Para possibilitar a descrição de valores dessa natureza, a linguagem Haskell dispõe de um construtor denominado ***tupla***. Podemos definir uma tupla como um agregado de dados, que possui quantidade pré-estabelecida de componentes (dois ou mais), e onde cada componente da tupla pode ser de um tipo diferente (primitivo ou não).

10.2. DEFINIÇÃO

A representação de uma tupla é feita com a seguinte sintaxe:

(t1, t2, t3, ..., tn)

Onde cada ***ti*** é um termo da tupla.

Se ***Ti*** é o tipo de ***ti***, então o universo de uma tupla é dado por:

TT = T1 x T2 x ... x Tn

Sabendo-se que os conjuntos bases dos tipos ***Ti*** são ordenados, também os elementos de ***TT*** o serão.

Exemplos de tuplas:

	Intenção	Representação
1	uma data	(15, 05, 2000)
2	uma posição no tabuleiro de xadrez	(3, 8)
3	uma pessoa	("Maria Aparecida", "solteira", (3, 02, 1970))
4	um ponto no espaço	(3.0, 5.2, 34.5)
5	uma carta de baralho	(7, "espada")
6	validação de dados	(True, 3)
7	validação de dados	(False, div 5 2)
8	uma tupla formada por expressões	(x/=0 && y /= 0, (x > y, *if* x*y /= 0 *then if* x > y *then* div x y *else* div y x *else* 0))
9	uma tupla formada por expressões	(x, y, x + y)

10.3. COMPONDO TUPLAS

Os exemplos apresentados já parecem suficientes para que tenhamos entendido como descrever um valor do tipo tupla. Vamos então apenas comentar sobre os exemplos e ilustrar o uso de tuplas nas definições.

Uma tupla é um valor composto. Isto significa que devemos colocar entre parêntesis e separados por vírgulas os componentes deste valor. Cada componente é um valor, descrito diretamente ou através de expressões envolvendo operadores. Nos exemplos de 1 a 7, todos eles usam constantes. O exemplo 7 apresenta um dos valores descrito por uma expressão. No exemplo 3, um dos termos é uma outra tupla. Qualquer termo pode ser uma tupla. Um valor pode também ser paramétrico, como no exemplo 9, onde temos uma tupla de 3 termos, todos eles paramétricos, mas o terceiro depende dos dois primeiros. Podemos dizer que esta tupla descreve todas as triplas onde o terceiro termo pode ser obtido pela soma dos dois primeiros. No exemplo 8 podemos observar o uso de expressões condicionais. Observa-se aí também a descrição de um termo do tipo boolean, através de uma expressão relacional combinada com operadores lógicos. Vejamos agora alguns exemplos de uso de tuplas na descrição de valores.

Exemplo 1: Desejamos definir uma função, para que dados 2 valores, seja produzido uma tripla onde os dois primeiros termos são idênticos aos elementos fornecidos, em ordem inversa, e o terceiro termo seja igual à soma dos dois.

```
triplaS a b = ( b, a, a + b )
```

Exemplo 2: Vamos definir uma função que produza o quociente e o resto da divisão inteira de dois números.

```
divInt a b = ( q, r)
             where
                q = div a b
                r = rem a b
```

Exemplo 3: Voltemos à definição das raízes de uma equação do 2o. grau. Vimos anteriormente que como eram duas, precisávamos definir também duas funções. Agora podemos agrupá-las em uma única definição:

```
re2g a b c = ( x1, x2)
             where
                x1     = ((-b) + e)/ duploA
                x2     = ((-b) - e)/ duploA
                e      =  sqrt (b^2 - 4.0*a*c)
                duploA = 2.0 * a
```

Exemplo 4: Voltemos aos movimentos do cavalo no jogo de xadrez. Vamos definir uma função que produza a nova posição, usando o primeiro movimento válido segundo o que se discutiu em Resolvendo Problemas - Os Movimentos do Cavalo.

```
qPos x y = if f x2 && f y1
           then (x2,y1)
           else if f x1 && f y2
                then (x1,y2)
                else if g x1e && f y2
                     then (x1e,y2)
                     else if g x2e && f y1
                          then (x2e,y1)
                          else if g x2e && g y1e
                               then (x2e,y1e)
                               else if g x1e && g y2e
                                    then (x1e,y2e)
                                    else if f x1 && f y2e
                                         then (x1,y2e)
                                         else (0,0)
           where
             x1    = x + 1
             x2    = x + 2
             y1    = y + 1
             y2    = y + 2
             x1e   = x - 1
             x2e   = x - 2
             y1e   = y - 1
             y2e   = y - 2
             f w   = w <= 8
             g w   = w >= 1
```

Qual o valor de qPos 1 9 ? O que há de errado? Reescreva qPos de forma a contornar o problema encontrado.

10.4. SELECIONANDO TERMOS DE UMA TUPLA

Assim com precisamos compor uma tupla na descrição dos mapeamentos de uma função, também precisamos decompô-la. Quando uma função possui tuplas como parâmetros, para usar seus termos é necessário que se possa referenciá-los.

Ilustraremos aqui várias formas, utilizando a definição que soluciona o seguinte problema:

Desejamos determinar a distância entre dois pontos no plano.

1. Utilizando a cláusula ***where*** :

```
dist p1 p2 = sqrt (dx^2 + dy^2)
             where
                 dx      = x1 - x2
                 dy      = y1 - y2
                 (x1,y1) = p1
                 (x2,y2) = p2
```

Observe a elegância da definição da tupla (x1, y1). Por análise da definição, sabemos que o tipo de p1 e p2 é tupla de 2 termos. Quando submetemos a avaliação de uma instância, o sistema casa p1 com um par de valores e a partir daí pode determinar o valor de cada termo de nossa tupla (x1, y1).

```
? dist (0.0,0.0) (5.0,5.0)
-- o sistema casa p1 com (0.0,0.0)
-- e p2 com (5.0,5.0)
-- logo x1=0.0, y1=0.0, x2=5.0 e y2=5.0
7.07107
```

2. Construindo funções seletoras :

```
prim (x,y) = x
seg  (x,y) = y
dist p1 p2 = sqrt (dx^2 + dy^2)
             where
                 dx      = prim p1 - prim p2
                 dy      = seg p1 - seg p2
```

Com esta abordagem, podemos generalizar e construir funções seletoras para tuplas com uma quantidade de termos qualquer. É importante destacar que tuplas com quantidade diferente de termos precisam ter o seu próprio elenco de funções seletoras.

Em particular, para o caso acima, onde o número de termos é 2, o próprio HUGS já possui as funções seletoras, ***fst*** (primeiro) e ***snd*** (segundo). Poderíamos tê-las usado diretamente, mas preferimos ilustrar como se constrói. Podemos reescrever a solução usando-as.

```
dist p1 p2 = sqrt (dx^2 + dy^2)
             where
                 dx      = fst p1 - fst p2
                 dy      = snd p1 - snd p2
```

3. Explicitando os componentes:

```
dist (x1,y1) (x2,y2) = sqrt (dx^2 + dy^2)
                       where
                         dx      = x1 - x2
                         dy      = y1 - y2
```

Exercícios:

1. Escreva uma função que determina as distâncias entre três pontos no plano cartesiano, duas a duas. Tanto os pontos dados como entrada, como as distâncias calculadas devem ser representadas por tuplas. Na resposta, cada par de pontos deve ser exibido, seguido da distância existente entre eles.

2. Refaça o exemplo de cálculo de equações de 2º grau, exibindo apenas as raízes reais da equação.

3. Dados a sua data de nascimento e a data atual, informe qual é a sua idade.

11. VALIDAÇÃO DE DADOS

11.1. INTRODUÇÃO

Como sabemos, toda função tem um domínio e um contradomínio. Em Hugs, quando tentamos avaliar uma instância de uma definição, usando valores fora desse domínio, podemos receber como resposta mensagens nem sempre esclarecedoras. Quando se trata do usuário de nosso programa, esta situação se mostra mais indesejável ainda. Aqueles que constroem a definição podem discernir com mais facilidade a natureza dos problemas que ocorrem durante o seu desenvolvimento. O mesmo não se pode esperar de alguém que não tem conhecimento dos elementos internos de nossas definições.

Tipicamente os erros serão de duas naturezas. Pode ser que o tipo da instância esteja correto, mas nossa definição use alguma função primitiva que não se aplica ao valor da instância. Por exemplo, se temos a definição:

```
f x y z = div x y + z
```

e se a submetemos a uma instância onde ***y = 0***, teremos como resultado algo da seguinte natureza:

```
? f 5 0 3

Program error: {5 `div` 0}

?
```

Um outro tipo de problema ocorre quando o tipo de algum parâmetro da nossa instância não casa com o tipo inferido pelo Hugs. Por exemplo, se usamos um valor do tipo ***Float*** para ***y***, obtemos:

```
? f 5 1.0 3
ERROR: Type error in
application
*** expression     : f 5 1.0 3
*** term           : 1.0
*** type           : Float
*** does not match : Int
```

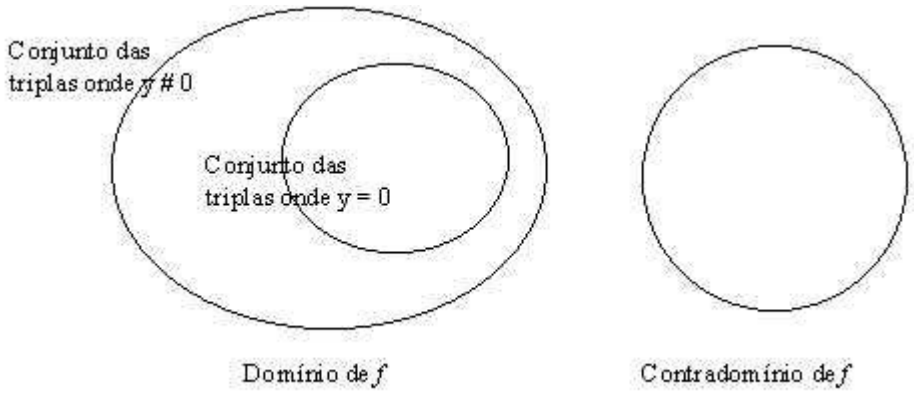
Nesta seção trataremos do primeiro caso, deixando o segundo para outra oportunidade.

11.2. CARACTERIZANDO A SITUAÇÃO

Voltemos à definição anteriormente apresentada:

```
f x y z = div x y + z
```

Neste caso, o tipo inferido pela linguagem para os dois primeiros parâmetros é o tipo ***Integer***, visto que o operador ***div*** só se aplica a valores deste tipo. Portanto o universo será formado por todos os possíveis ternos de valores onde os dois primeiros são do tipo ***Integer*** e o último, do tipo obtido pela união dos tipos ***Float*** e ***Integer.*** Chamemo-lo de ***T***. Entretanto, o domínio de nossa função não é exatamente o conjunto de constantes de ***T***, posto que a função não está definida para as constantes de ***T*** cujo segundo elemento é nulo.



Desejamos construir uma função que seja uma extensão da função original e ao mesmo tempo lhe sirva como uma casca de proteção contra as violações de domínio. Precisamos escolher um contradomínio da função estendida (***fx***). Um candidato natural é o contradomínio da função original (***f***).

E a imagem de nossa função, o que podemos dizer dela? Será que ela é igual ao contradomínio? Ou será que temos um subconjunto para o qual não exista um mapeamento?

11.3. CASO GERAL (IMAGEM IDÊNTICA AO CONTRADOMÍNIO)

No caso geral, podemos encontrar duas situações: ou o contradomínio é idêntico à imagem, ou a determinação dos elementos que não pertencem à imagem pode não ser fácil. Com isto não dispomos de valores no contradomínio que possam ser utilizados para mapearmos os valores que violam o domínio de ***f***. Neste caso podemos estender o contradomínio (CD) de tal modo que o novo escolhido incorpore um valor que será a imagem da extensão de ***f***.

Uma solução geral bastante conveniente é construir um novo contradomínio (NCD) para ***fx*** (extensão de ***f***), formado por pares onde o primeiro elemento é do tipo boolean e o segundo do contradomínio de ***f***. Temos então a seguinte estrutura:

$$\mathbf{NCD}(fx) = \mathbf{(boolean, CD}(f)\mathbf{)}$$

Para valores de ***x*** no domínio de ***f***, teremos:

$$fx\ x = \mathbf{(True,}\ f\ x\ \mathbf{)}$$

Para os valores de ***x*** fora do domínio teremos:

$$fx\ x = (\text{False}, k)$$

onde ***k*** é um valor qualquer pertencente ao contradomínio de ***f***.

Para o nosso exemplo inicial teríamos então:

```
fx x y z = if invalido x y z
             then (False,0)
             else (True, f x y z)
             where
               invalido x y z = ...
f x y z = div x y + z
```

É como se estivéssemos criando uma casca protetora para a ***f.***

11.4. FUNÇÕES QUE APRESENTAM PROBLEMAS EM VÁRIOS PARÂMETROS

Quando uma função possui vários parâmetros, pode ocorrer que mais de um deles dêem origem à questão que aqui levantamos. Quando isso ocorre, pode ser relevante caracterizar a situação apropriadamente. Neste caso podemos usar um conjunto mais variado de constantes do que as constantes booleanas, permitindo que possamos associar com cada erro uma constante diferente.

Podemos tomar como exemplo o problema do movimento dos cavalos no jogo de xadrez, especificamente a solução genérica que produzimos com a função

mov m x y

onde ***m*** é o número do movimento, ***x*** a linha atual e ***y*** a coluna atual.

Os três parâmetros são válidos apenas para o intervalo [1, 8]. Portanto ***mov*** não está definida para os valores pertencentes ao subconjunto do universo formado por todas as triplas onde pelo menos um dos elementos não pertence ao intervalo [1, 8]. Por outro lado, o contradomínio é o conjunto booleano, portanto só possui duas constantes e ambas já estão comprometidas. Se quisermos distinguir os três tipos de violação do domínio (movimento inválido, posição inválida, ambos inválidos) precisaremos usar um conjunto com pelo menos quatro constantes.

Podemos então definir a função “movx” da seguinte forma:

```
movx m x y = if not validom
             then if not validop
                  then (3, False)
                  else (1, False)
             else if not validop
                  then (2, False)
                  else (0, mov m x y)
```

11.5. CASO PARTICULAR (IMAGEM DIFERENTE DO CONTRADOMÍNIO)

Suponha que existe pelo menos um elemento ***k*** que não pertence à imagem, ou seja, a imagem está contida no contradomínio. Podemos construir uma extensão de ***f*** de tal forma que os elementos que não pertençam ao domínio sejam mapeados neste ***k*** e os demais sejam mapeados diretamente pela ***f***. Quando existe tal ***k***, nosso problema está resolvido. Basta que o usuário saiba que, quando a avaliação resultar em ***k***, significa que a função não se aplica para aquela instância.

Voltemos à função **f** introduzida anteriormente. Podemos, usando a proposta desta seção, reescrever a sua definição, da seguinte forma:

```
fx x y z = if invalido x y z
             then k
             else f x y z
             where
               k = ...
               invalido x y z = ...
f x y z  = div x y + z
```

Infelizmente, para esse caso, o ***k*** não existe (prove!).

Voltemos ao movimento do cavalo. Nesse caso, especificamente, porque o contradomínio original é o conjunto booleano, poderíamos ter tomado outro caminho. Poderíamos usar números negativos para indicar os três tipos de violação do domínio, a constante inteira ***0*** para representar ***False*** e a constante inteira ***1*** para representar ***True***, eliminando com isso a necessidade de um novo domínio formado por pares. Vejamos como fica esta definição:

```
movx m x y = if not validom
             then if not validop
                  then (-3)
                  else (-1)
             else if not validop
                  then (-2)
                  else if mov m x y
                       then 1
                       else 0
```

11.6. UM EXEMPLO - RAIZES DE UMA EQUAÇÃO DO 2O. GRAU

Voltemos ao problema de determinar as raízes de uma equação do segundo grau. Já sabemos que elas são duas e que podemos fazer uma única função para descrevê-las, usando tupla.

Sabemos ainda que o universo definido pelos tipos dos três parâmetros (a, b, c), é maior que o domínio da função. Ou seja, a função não está definida para instâncias de a, b e c, onde se verifica a seguinte desigualdade:

```
(b^2) + (4 * a * c) < 0
```

Precisamos, portanto de uma função estendida.

```
re2gx a b c = (not (delta < 0), if delta < 0
                                then (0,0)
                                else (x1, x2))
              where
                 delta = (b ^ 2) + (4 * a * c)
                 x1    = ((-b) + sqrt delta) / (2 * a)
                 x2    = ((-b) - sqrt delta) / (2 * a)
```

Ou, de outra forma:

```
re2gx a b c = if delta < 0
              then (False, (0, 0))
              else (True, (x1, x2))
              where
                  delta = (b ^ 2) + (4 * a * c)
                  x1    = ((-b) + sqrt delta) / (2 * a)
                  x2    = ((-b) - sqrt delta) / (2 * a)
```

Exercícios

1. Explique por que não existe o valor k que apóie a definição da função ***fx***;
2. Faça uma função estendida para f x y = 1/(x+ y)
3. Forneça dois exemplos de funções para as quais é possível criar uma função estendida na qual se possa usar constantes que não pertencem garantidamente à imagem da função original

12. LISTAS

12.1. INTRODUÇÃO

A maioria dos itens de informação de nosso interesse está agrupada, dando origem a um conceito muito importante para a resolução de problemas, o agrupamento. Freqüentemente estamos interessados em manipular esses agrupamentos para extrair informações, definir novos itens ou avaliar propriedades desses agrupamentos.

Tratamos anteriormente das tuplas, que são agrupamentos de tamanho predefinido e heterogêneo, ou seja, cada elemento que participa do agrupamento pode ser de um tipo diferente. Agora estamos interessados em explorar outro tipo de agregação, as listas. Este novo tipo, em HUGS, caracteriza-se por agregar quantidades variáveis de elementos desde que todos eles sejam de um mesmo tipo.

Vivemos cercados de listas. Elas estão em qualquer lugar onde precisamos registrar e processar dados. Vejamos alguns exemplos:

1. Lista de números pares;
2. Lista dos livros lidos por uma pessoa;
3. Lista dos amigos que aniversariam em um dado mês;
4. Lista dos presidentes corruptos;
5. Lista dos vereadores decentes;
6. Lista das farmácias enganadoras;
7. Lista das disciplinas que já cursei;
8. Lista dos lugares que visitei;
9. Lista dos números feios;
10. Lista dos números primos;
11. Lista das posições para as quais um cavalo pode se deslocar;
12. Lista das palavras de um texto;
13. Lista dos erros provocados pelo Windows;
14. Lista dos prêmios Nobel ganhos pelo Bertrand Russel;
15. Lista dos títulos conquistados pelo Nacional Futebol Clube;
16. Listas dos *funks* compostos pelo Noel Rosa.

Destas, algumas são vazias, outras são finitas e algumas infinitas.

12.2. CONCEITOS BÁSICOS: Uma lista é uma seqüência de zero ou mais elementos de um mesmo tipo.

Entende-se por seqüência uma quantidade qualquer de itens dispostos linearmente.

Podemos representar uma lista pela enumeração dos seus elementos, separados por vírgulas e cercados por colchetes.

[***e1***, ***e2***, .., ***en*****]**

Por exemplo:

1. [ ]
2. [1,3,5,7,9]
3. ['a', 'e', 'i', 'o', 'u']
4. [(22,04,1500), (07,09,1822), (31,03,1964)]
5. [[1,2,5,10], [1,11], [1,2,3,4,6,12], [1,13], [1,2,7,14], [1,3,5,15]]

É importante ressaltar que em uma lista podemos falar do primeiro elemento, do quinto, do enésimo, ou do último. Ou seja, há uma correspondência direta entre os números naturais e a posição dos elementos de uma lista.

Este último fato nos lembra de um equívoco freqüente, que queremos esclarecer de saída. A ordem que se adota em listas, por ser baseada nos números naturais, começa do zero. Ou seja, o primeiro elemento de uma lista tem o número de ordem igual a zero.

Por exemplo, na relação acima apresentada, a primeira lista é vazia. Na lista do item 4 o elemento de ordem 1 é a tupla (07, 09,1822) e na lista do item 5, o elemento de ordem zero (0) é a lista [1,2,5,10].

Quanto ao tipo, podemos dizer que a segunda lista é uma lista de números, a terceira uma lista de caracteres, a quarta é uma lista de triplas de números e a quinta é uma lista de listas de números. Qual será o tipo da primeira lista?

Uma lista vazia é de natureza polimórfica, isto é, seu tipo depende do contexto em que seja utilizada, como veremos em momento oportuno.

12.3. FORMAS ALTERNATIVAS PARA DEFINIÇÃO DE LISTAS

Além da forma básica, acima apresentada, também conhecida como enumeração, onde explicitamos todos os elementos, dispomos ainda das seguintes maneiras: definição por intervalo, definição por progressão aritmética e definição por compreensão. As duas primeiras formas são apresentadas a seguir e a terceira, posteriormente.

Definição por Intervalo

De uma forma geral, podemos definir uma lista explicitando os limites inferior e superior de um conjunto conhecido, que possua uma relação de ordem entre os elementos, no seguinte formato:

[<limite inferior> **..** <limite superior>**]**

Exemplos: abaixo listamos algumas submissões de listas definidas por intervalo e as correspondentes respostas do sistema HUGS.

```
1 Prelude> [1..5]
  [1,2,3,4,5]
2 Prelude> [-2..2]
  [-2,-1,0,1,2]
3 Prelude> [10..5]
  []
4 Prelude> [-5.0..5.0]
  [-5.0,-4.0,-3.0,-2.0,-1.0,0.0,1.0,2.0,3.0,4.0,5.0]
5 Prelude> [-1..-5]
  ERROR: Undefined variable "..-"
6 Prelude> [-1..(-5)]
  []
7 Prelude> [1.6..10.0]
  [1.6,2.6,3.6,4.6,5.6,6.6,7.6,8.6,9.6]
8 Prelude> [1.5..10.0]
  [1.5,2.5,3.5,4.5,5.5,6.5,7.5,8.5,9.5,10.5]
```

Podemos observar aqui algumas aparentes surpresas: no exemplo 4 podemos observar que o HUGS considera sempre o intervalo de uma unidade entre os elementos da lista definida. No exemplo 5, temos uma questão de ambigüidade, 0 sinal de menos logo após os dois pontos não é entendido e o sistema sinaliza erro. O exemplo 6 contorna essa dificuldade. Deixo ao leitor buscar explicação para o que ocorre com os exemplos 7 e 8.

Definição por Progressão Aritmética

Podemos definir qualquer progressão aritmética por uma lista utilizando a seguinte notação:

[<1o. termo>, <2o. termo> **..** <limite superior>**]**

Exemplos: observemos as definições a seguir e as respectivas respostas do sistema HUGS. Podemos perceber que dependendo do segundo termo da definição o sistema interpreta a P.A. como crescente ou decrescente.

```
1 Prelude> [1,2..6]
  [1,2,3,4,5,6]
2 Prelude> [-5,2..5]
  [-5,2]
3 Prelude> [-5,2..15]
  [-5,2,9]
4 Prelude> [-5,2..16]
  [-5,2,9,16]
5 Prelude> [1,1.1 .. 2]
  [1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8,1.9,2.0]
6 Prelude> [6,5..0]
  [6,5,4,3,2,1,0]
7 Prelude> [5,6..5]
  [5]
8 Prelude> [3,7..1]
  []
```

12.4. OPERAÇÕES BÁSICAS

As listas, como já dissemos, são elementos da linguagem que podemos utilizar para descrever valores estruturados. Como nos demais tipos da linguagem, valores descritos por listas podem e devem ser usados na descrição de novos valores através da utilização de operações definidas sobre eles. A seguir são apresentadas algumas dessas operações.

elem : avalia se um determinado elemento é membro de uma lista.

elem <elemento> <lista>

Exemplos:

```
Main> elem 5 [-5..15]
True
Main> elem (-10) [-5..15]
False
Main> elem 20 [-5..15]
False
Main> elem 15 [-5..15]
True
```

length : descreve o tamanho de uma lista.

length <lista>

Exemplos:

```
Prelude>length [1..100]
100
(2528 reductions, 3442 cells)
Prelude>length [1..100] + length [1..100]
200
(5050 reductions, 6873 cells)
Prelude> a + a where a = length [1..100]
200
(2530 reductions, 3442 cells)
```

indexação : podemos descrever cada termo de uma lista como um novo valor. Para tanto basta indicarmos a posição do elemento dentro da lista, considerando que o primeiro elemento tem a ordem zero.

<lista>**!!**<índice>

Exemplos:

```
1 Prelude> [3,4,6,74,45]!!0
  3
2 Prelude> [3,4,6,74,45]!!4
  45
3 Prelude> [1,5..20]!!3
  13
4 Prelude>  [[[1],[2,3,4]]]!!0!!1!!2
  4
5 Prelude> [3,4,6,74,45]!!3
  74
```

```
6 Prelude> [3,4,6,74,45]!!3 + [2,3]!!0
  76
7 Prelude> [3,4,6,74,45]!!(0+1)
  4
8 Prelude> [3,4,6,74,45]!!(i+j) where i=3; j=0
  74
9 Prelude> [84,76,23,124,763,23]!!([0,1,2,3]!!2)
  23
10 Prelude> [1,5..20]!!5
  Program error: Prelude.!!: index too large
```

concat : descreve uma nova lista obtida pela concatenação de uma lista de listas.

concat <lista de listas>

Exemplos:

```
1 Prelude>concat [[1..5],[1,10..100],[9]]
 [1,2,3,4,5,1,10,19,28,37,46,55,64,73,82,91,100,9]
(473 reductions, 747 cells)
2 Prelude>concat [[1]]
  [1]
  (25 reductions, 35 cells)
3 Prelude> length (concat [[1..5],[1,10..100],[9]])
  18
  (451 reductions, 638 cells)
4 Prelude> concat [[length [10,87..500]], [length
[10,87,500]]]
  [7,3]
  (228 reductions, 309 cells)
5 Prelude> length (concat [[length [10,87..500]],
[length    [10,87,500]]])
  2
  (30 reductions, 35 cells)
```

Pode também ser usado com a sintaxe infixada usando o operador **++**

Exemplos:

```
1 Prelude> [1] ++ [1]
  [1,1]
  (33 reductions, 48 cells)
2 Prelude> [1] ++ []
  [1]
  (23 reductions, 33 cells)
3 Prelude> [ ] ++ [ ]
  ERROR: Cannot find "show" function for:
  *** Expression : [] ++ []
  *** Of type    : [a]
4 Prelude>  [1..5] ++ [1,10..100] ++ [9]
  [1,2,3,4,5,1,10,19,28,37,46,55,64,73,82,91,100,9]
  (467 reductions, 738 cells)
```

construtor : descreve uma nova lista onde o primeiro termo é um elemento dado e os demais são os componentes de uma lista também dada.

<elemento> **:** <lista>

Exemplos:

```
1 Prelude> 3: [4,5,6]
  [3,4,5,6]
2 Prelude> [3]:[[4,5,6]]
  [[3],[4,5,6]]
3 Prelude> [3: [4,5,6]]
  [[3,4,5,6]]
4 Prelude> [3: [4,5,6]]!!0
  [3,4,5,6]
5 Prelude> [3: [4,5,6]]!!0!!3
  6
```

head : descreve o primeiro elemento de uma lista.

tail : descreve uma sublista contendo todos os elementos, exceto o primeiro elemento de uma lista dada.

Teorema importante:

head xs : (tail xs) = xs

last : descreve o último elemento de uma lista..

init : descreve uma sublista contendo todos os elementos, exceto o último elemento de uma lista dada.

Teorema importante:

init xs ++ [last xs] = xs

Importante: As funções head, last, init e tail não possuem em seu domínio a lista vazia. Portanto ela não deve ser instância de avaliação para qualquer uma delas.

<elemento> **:** <lista>

Exemplos:

```
1 Hugs> head [1,2,3,4,5,6]
  1
2 Hugs> tail [1,2,3,4,5,6]
  [2,3,4,5,6]
3 Hugs> tail [1,2,3,4,5,6]
  [2,3,4,5,6]
4 Hugs> last [1,2,3,4,5,6]
  6
5 Hugs> init [1,2,3,4,5,6]
  [1,2,3,4,5]
6 Hugs> x == (head x : tail x) where x = [10..15]
  True
7 Hugs> init [10..15] ++ tail [10..15]
  [10,11,12,13,14,11,12,13,14,15]
```

12.5. DEFINIÇÃO POR COMPREENSÃO

Uma lista pode ser descrita através da enumeração de seus elementos, como já vimos através de vários exemplos. Esta forma é também denominada de definição por extensão, visto que todos os componentes precisam ser explicitados.

Podemos também descrever listas através das condições que um elemento deve satisfazer para pertencer a ela. Em outras palavras, queremos descrever uma lista através de uma intenção. Esta forma é análoga à que já conhecemos para descrição de conjuntos. Por exemplo, é usual escrever a notação abaixo para descrever o conjunto formado pelo quadrado dos números naturais menores que 10.

P = {quadrado de x | x é natural e é menor que 10}

ou ainda mais formalmente,

P = { x^2 | x pertence a N e x <10}

Podemos observar, no lado direito da definição, que ela é formada por duas partes. A primeira é uma expressão que descreve os elementos, usando para isso termos variáveis que satisfazem condições de pertinência estabelecidas pela segunda parte.

expressão	x^2
variável	x
pertinência	x pertence a N e x <10

Em nosso caso, não estamos interessados em descrever conjuntos e sim listas. E isso tem algumas implicações práticas. Por exemplo, em um conjunto a ordem dos elementos é irrelevante, para listas não. É bom lembrar ainda que em uma lista, o mesmo valor pode ocorrer varias vezes, em diferentes posições.

A sintaxe que usaremos é a seguinte:

[<expressão> **|** <pertinência> **]**

Onde:

<expressão> - expressões usuais em Haskell para definição de valores, inclusive condicionais.

<pertinência> - descrição dos elementos a serem considerados para avaliação da <expressão>.

A pertinência é formada por uma seqüência de qualificadores de dois tipos de construção: os geradores e os predicativos. Os geradores descrevem uma lista de onde se originam os elementos a serem considerados, usando a sintaxe:

```
<termo> <- <lista>
```

Por exemplo, vejamos a descrição da lista dos quadrados dos números menores que 5.

```
Prelude> [x^2| x<-[0..4]]
[0,1,4,9,16]
```

Os predicativos são expressões descrevendo valores booleanos, envolvendo termos já definidos anteriormente (inclusive por geradores).

Vejamos o exemplo a seguir, onde descrevemos uma sublista de números ímpares, tendo como origem de geração uma lista definida por uma Progressão Aritmética.

```
Prelude> [x | x<-[1,4..100],odd x]
[1,7,13,19,25,31,37,43,49,55,61,67,73,79,85,91,97]
```

É importante destacar que as expressões de pertinência são avaliadas da esquerda para direita.

Por exemplo, se na expressão acima, primeiro colocássemos a expressão booleana "odd x", o sistema acusaria um erro, visto que ao tentar a avaliar "odd x", a variável "x" ainda não estaria definida.

```
Prelude> [x | odd x, x<-[1,4..100]]
ERROR: Undefined variable "x"
```

Isto só ocorre porque o sistema usa uma ordem pré-definida!

Vejamos como usar este novo conceito na escrita de novos scripts. A seguir apresentamos a definição de três novas funções. A primeira, **slpares**, define uma sublista formada pelos quadrados dos elementos pares de uma lista dada. A segunda, **lmenor**, define uma sublista formada pelos elementos de uma dada lista, que são menores que um elemento fornecido. A terceira, **pmaioresk**, ilustra o uso da cláusula **if-then-else** para a geração de elementos de uma lista.

```
-- Dada uma lista 'xs' defina uma sublista formada
-- pelo quadrado dos elementos que são pares
--
slpares xs = [x^2 | x<- xs, even x]

Main> :l preparalista.txt
Reading file "preparalista.txt":

Hugs session for:
E:\HUGS98\lib\Prelude.hs
preparalista.txt

Main> slpares [1,4..50]
[16,100,256,484,784,1156,1600,2116]

Main> slpares  [34,67,99,23,12,3,67,99]
[1156,144]
```

```
--
-- Determinar a sublista de elementos menores que 'x'
-- em uma lista 'xs
--
lmenor x xs = [ y | y <-xs, y < x]

...

Main> lmenor 45 [1,5,6,86,34,76,12,34,86,99]
[1,5,6,34,12,34]

Main> lmenor 1 [1,5,6,86,34,76,12,34,86,99]
[]

Main> lmenor 100 [1,5,6,86,34,76,12,34,86,99]
[1,5,6,86,34,76,12,34,86,99]
```

```
--
-- Determinar a lista de elementos gerados condicionalmente
-- a uma constante dada k.

pmenoresk k xs = [ if x > k then x2 else x + 2 | x <-xs]

...

Main> pmenoresk 30 [1,5,6,86,34,76,12,34,86,99]
[1,25,36,88,36,78,144,36,88,101]
```

Quando mais de um gerador é utilizado, devemos levar em conta que para cada elemento do gerador mais a esquerda serão gerados todos os elementos dos geradores subseqüentes. Vejamos o exemplo a seguir onde se descreve uma lista de pontos do plano cartesiano, localizados no primeiro quadrante e delimitado pelas ordenadas 3 e 5.

```
--
-- Determinar a lista dos pontos do plano dentro da
-- regiao definida pela origem, a cordenada (eixo y)
--  5 e a abscissa (eixo x) 3.
--
pontos = [ (x,y) | x <- [0..3], y <- [0..5]]

...

Main> pontos
[(0,0),(0,1),(0,2),(0,3),(0,4),(0,5),
(1,0),(1,1),(1,2),(1,3),(1,4),(1,5),
(2,0),(2,1),(2,2),(2,3),(2,4),(2,5),
(3,0),(3,1),(3,2),(3,3),(3,4),(3,5)]
```

Entre dois geradores podemos usar predicativos, como se observa no exemplo a seguir.

```
--
-- Determinar a lista dos pontos do plano dentro da
-- regiao definida pela origem, a cordenada (eixo y)
-- 5 e a abscissa (eixo x) 3. Considere apenas os
-- pontos onde x é impar e y par.
--
pontos1 = [ (x,y) | x <- [0..3], odd x, y <- [0..5], even y ]
...

Main> pontos1
[(1,0),(1,2),(1,4),(3,0),(3,2),(3,4)]
```

12.6. DEFINIÇÃO POR COMPREENSÃO - EXPLORANDO DETALHES

Vamos explorar um problema onde exista mais riqueza de detalhes quanto ao uso de predicativos.

Determinar os pares de valores, onde:

- o primeiro é múltiplo do segundo;
- o primeiro é dado pelos elementos impares de uma P.A de primeiro termo igual a 1, a razão igual 3 e o ultimo termo menor ou igual a 100;
- o segundo termo é dado pelos elementos de uma P.A de primeiro termo igual a 1, a razão igual 4 e o ultimo termo menor ou igual a 50;
- um dos dois é diferente da unidade;
- os dois termos são diferentes.

```
--
-- A função paresE, traduz literalmente o enunciado.
-- As P.A.'s são realizadas diretamente pelo mecanismo
-- para definição de P.A. embutido na linguagem.
-- Os predicativos foram inseridos na posição em que
-- suas variáveis já estão instanciadas.

--
paresE1  = [ (x,y) | x <- [1,4..100],
                     odd x,
                     y <- [1,5..50],
                     (x/=1 || y/= 1),
                     x/=y,
                     mod x y == 0]
...
Main> paresE1
[(7,1),(13,1),(19,1),(25,1),(25,5),(31,1),(37,1),
(43,1),(49,1),(55,1),(55,5),(61,1),(67,1),(73,1),
(79,1),(85,1),  (85,5),(85,17),(91,1),(91,13),(97,1)]
```

Vejamos algumas observações sobre a solução acima apresentada:

- o predicado odd poderia ser colocado em qualquer lugar mais a frente, entretanto o desempenho cairia, visto que iríamos produzir valores desnecessários para "y";

- a expressão **(x/=1 || y/= 1)** é desnecessária visto que só será falsa quando ambos, x e y, forem iguais a 1, mas nesse caso eles serão iguais e portanto falsificariam a expressão a seguir **x/=y**;

- podemos reescrever a expressão **x <- [1,4..100]** de tal maneira que gere apenas valores ímpares e assim descartar a expressão **odd x**. Para tanto basta mudar a P.A. para **[1,7..100];**

- pode-se observar que os valores de **y** que interessam são sempre menores que os de **x** (já que **y** é divisor de **x**). Portanto, a segunda P.A poderia ser substituída por **[1,5..(x-1)]**. Acontece que agora poderemos geram valores para **y** maiores que 50 e isto não interessa. O que fazer? Que tal substituí-la por:

[1, 5..(if x < 50 then (x-1) else 50)]

Eis então uma nova versão para nossa função:

```
paresE2  = [ (x,y) | x <- [1,7..100],
                     y <- [1,5..(if x < 50
                                 then (x-1)
                                 else 50)],
                     x/=y,
                     mod x y == 0]
```

Podemos agora refletir sobre uma possível generalização para a nossa função, considerando-se duas listas quaisquer. Neste caso, o esforço realizado para melhorar o

desempenho seria em vão porque não conhecemos a priori a natureza das duas listas. Nossa função poderia ser:

```
paresE3 xs ys = [ (x,y) | x <- xs,
                          odd x,
                          y <- ys,
                          x/=y,
                          mod x y == 0]
```

Apenas a expressão **(x/=1 || y/= 1)** poderia ser eliminada. O objetivo de nossa versão original poderia ser obtido pelo uso da nova função aplicada às listas específicas. Conforme se observa a seguir:

```
Main> paresE3 [1,4..100] [1,5..50]
[(7,1),(13,1),(19,1),(25,1),(25,5),(31,1),(37,1),(43,1),
(49,1),(55,1),(55,5),(61,1),(67,1),(73,1),(79,1),(85,1),
(85,5),(85,17),(91,1),(91,13),(97,1)]
(16878 reductions, 23992 cells)
```

Agora podemos ainda inspecionar o que de fato acontece com o desempenho das três versões.

```
Main> paresE1
[(7,1),(13,1),(19,1),(25,1),(25,5),(31,1),(37,1),(43,1),
(49,1),(55,1),(55,5),(61,1),(67,1),(73,1),(79,1),(85,1),
(85,5),(85,17),(91,1),(91,13),(97,1)]
(22894 reductions, 32210 cells)

Main> paresE2
[(7,1),(13,1),(19,1),(25,1),(25,5),(31,1),(37,1),(43,1),
(49,1),(55,1),(55,5),(61,1),(67,1),(73,1),(79,1),(85,1),
(85,5),(85,17),(91,1),(91,13),(97,1)]
(15124 reductions, 22014 cells)
```

Coloquemos os valores obtidos em uma tabela.

versão	reduções	células
paresE1	22894	32210
paresE2	15124	22014
paresE3	16878	23992

Aqui podemos constatar que a versão paresE3, bastante similar a paresE1, a menos da generalização, possui eficiência bem superior. Lembre-se que a única diferença é a eliminação da comparação dos dois termos com o valor 1.

Para ver o que está acontecendo, vamos construir ainda uma outra versão, similar a paresE1, eliminando a comparação citada e descrevendo as P.A.'s fora dos geradores,

assimilando-se assim à versão paresE3. Veja que com isso a nova versão, que obtivemos partindo de paresE1, possui eficiência similar a paresE3. Confirma-se então a observação de que a definição de listas dentro dos geradores produz perdas.

```
paresE4  = [ (x,y) | x <- xs,
                     odd x,
                     y <- ys,
                     x/=y,
                     mod x y == 0]
           where
             xs = [1,4..100]
             ys = [1,5..50]
...
main> paresE4
[(7,1),(13,1),(19,1),(25,1),(25,5),(31,1),(37,1),(43,1),
(49,1),(55,1),(55,5),(61,1),(67,1),(73,1),(79,1),(85,1),
(85,5),(85,17),(91,1),(91,13),(97,1)]
(16878 reductions, 23992 cells)
```

12.7. OPERAÇÕES PARA DETERMINAR SUBLISTAS

Existem algumas operações predefinidas no Haskell para descrevermos sublistas de uma lista dada. Nada que não possa ser feito com o que já apresentamos até aqui. Entretanto o seu uso pode ajudar na prática do reuso e contribuir bastante para a clareza de um programa.

Grupo I - Considerando um tamanho especificado

take k xs - define uma lista com os k primeiros elementos de uma lista xs.

drop k xs - define uma lista com os elementos de xs, a partir do elemento seguinte aos k primeiros.

Vejamos a ilustração a seguir.

```
Prelude> take 3 [0..10]
[0,1,2]
(156 reductions, 221 cells)
Prelude> drop 3 [0..10]
[3,4,5,6,7,8,9,10]
(358 reductions, 513 cells)
Prelude> [xs!!i | i <-[0..(k - 1)]]
         where xs = [0..10]; k = 3
[0,1,2]
(249 reductions, 336 cells)
Prelude> [xs!!i | i <-[k..length xs - 1]]
         where xs = [0..10]; k = 3
[3,4,5,6,7,8,9,10]
(1591 reductions, 2072 cells)
Prelude> (take 3 [1..10] ++ drop 3 [1..10]) == [1..10]
True
(658 reductions, 980 cells)
```

Na verdade podemos escrever uma descrição geral para take e drop usando listas por compreensão. Além disso, sabemos que a concatenação de take e drop para um certo k, aplicado à uma lista xs, é igual à própria lista.

```
take k xs = [xs!!i | i <-[0..(k - 1)]]

drop k xs = [xs!!i | i <-[k..length xs - 1]]

vale o seguinte teorema:
               take k xs ++ drop k xs = xs
```

Grupo II - Considerando a satisfação de um predicado *pred*. Vale lembrar que um predicado é uma função cujo valor resultante é do tipo boolean.

takeWhile pred xs - define uma lista com os primeiros elementos de uma lista xs satisfazendo o predicado pred.

dropWhile pred xs - define uma lista com os elementos de xs, a partir do primeiro elemento que não satisfaz o predicado pred.

Vejamos a ilustração a seguir.

```
 Prelude> takeWhile even [1..10]
[]

Prelude> takeWhile odd [1..10]
[1]

Prelude> dropWhile odd [1..10]
[2,3,4,5,6,7,8,9,10]

Prelude> dropWhile even [1..10]
[1,2,3,4,5,6,7,8,9,10]

Prelude> takeWhile (<5) [1..10]
[1,2,3,4]

Prelude> dropWhile (<5) [1..10]
[5,6,7,8,9,10]

Prelude> takeWhile (<5) [1..10] ++
         dropWhile (<5) [1..10] == [1..10]
True
```

13. RESOLVENDO PROBLEMAS COM LISTAS

Neste capítulo desenvolveremos a solução para alguns problemas mais complexos. A intenção é apresentar o uso de estratégias de propósito geral que podem ser úteis na resolução de novos problemas. Discutimos também algumas questões relativas ao desempenho das soluções.

13.1 PROBLEMA 1 : *Dada uma lista, determine o seu maior elemento*.

Comecemos por definir, usando a linguagem de conjuntos, quem é este elemento.

Dizemos que k é o maior elemento de um conjunto C, se e somente se, o subconjunto de C formado por todos os elementos maiores que k é vazio.

Em linguagem de lista isto equivale a dizer que se C é uma lista, a sublista de C formada pelos caras de C maiores que k é vazia.

Vejamos como fica a codificação em Haskell.

```
--
-- definindo uma função para determinar, em uma lista,
-- a sublista dos elementos maiores que um dado x
--
maiores x xs = [ y | y <- xs, y > x]
--
--
-- Em listas, podemos ter elementos repetidos, e em particular
-- podemos ter vários exemplares do maior elemento.
-- Chamemos esses caras de os "maiorais da lista".
-- Vamos construir uma função para descrevê-los.
--
maiorais xs = [ k | k <- xs, maiores k xs == [] ]
--
-- Como eles são todos idênticos podemos tomar o primeiro deles
-- como solução de nosso problema.
--
maximo xs = head (maiorais xs)
--
```

13.2 PROBLEMA 2: *Dada uma lista, verifique se ela é não decrescente.*

Como aquecimento, vamos considerar listas de números e a noção usual de ordem não decrescente. Antes de programar, vamos resgatar a definição de seqüências não decrescentes.

Definição: Uma seqüência S está em ordem não decrescente se e somente se qualquer um de seus elementos é menor ou igual aos seus sucessores. Em outras palavras, podemos dizer que a coleção de elementos de S que são maiores que seus sucessores é vazia.

```
--
--  lista dos maiores que os sucessores
--
lms1 xs = [ xs!!i | i <-[0..length xs - 2],
                    j <-[i+1..length xs - 1],
                    xs!!i > xs!!j]
--
-- a seqüência está ordenada se a lista dos elementos
-- que são maiores que algum sucessor é vazia
--
ord1 xs = (lms1 xs) == []
...

Main> ord1 [1..10]
True
(10801 reductions, 13891 cells)
Main> ord1 [10,9..1]
False
(405 reductions, 571 cells)
```

13.3. DISCUTINDO EFICIÊNCIA: Observando a avaliação da função **ord1**, apresentada na seção anterior, para uma lista já ordenada, a lista [1,2,3,4,5,6,7,8,9,10], constatamos um número muito grande de reduções. Convém questionar os motivos.

```
Main> ord1 [1..10]
True
(10801 reductions, 13891 cells)
Main> ord1 [10,9..1]
False
(405 reductions, 571 cells)
```

Em geral os motivos para um baixo desempenho são de duas naturezas: exploração inadequada das propriedades do problema e escolha inadequada dos mecanismos da linguagem. A seguir fazemos uma pequena exploração desses dois aspectos.

13.3.1. EXPLORANDO PROPRIEDADES DO PROBLEMA - Analisando a nossa definição anterior constatamos que ela diz mais do que precisamos. Ela avalia cada elemento com respeito a todos sucessores. Na verdade, nossa definição pode ser melhorada. Basta saber que cada elemento tem que ser menor ou igual ao seu sucessor imediato. Vamos reescrever a nossa definição:

Definição : Uma seqüência S está em ordem não decrescente se e somente se qualquer um de seus elementos é menor ou igual ao seu **sucessor imediato**. Em outras palavras, podemos dizer que a coleção de elementos de S que são maiores que seus **sucessores imediatos** é vazia.

Vejamos então a implementação em Haskell e sua aplicação às mesmas instâncias do problema:

```
--
-- lista dos elementos maiores que o sucessor imediato
--

lms2 xs = [ xs!!i | i <-[0..length xs - 2],
                    xs!!i > xs!!(i+1)]
--
ord2 xs = (lms2 xs) == []
...

Main> ord2 [1..10]
True
(2236 reductions, 2885 cells)
Main> ord2 [10,9..1]
False
(314 reductions, 449 cells)
```

Podemos observar uma queda no número de *reduções* da ordem de 79%!!! Para o pior caso, ou seja, quando todos os elementos satisfazem a propriedade estabelecida. No melhor caso, onde o primeiro já não satisfaz a propriedade, a queda foi da ordem de 22%. Na utilização de células, os ganhos foram da mesma ordem.

13.3.2. EXPLORANDO OS MECANISMOS DA LINGUAGEM - Uma outra investida que podemos fazer é com respeito ao uso adequado dos mecanismos da linguagem. As soluções acima apresentadas processam as listas através de índices, ou seja, fazem um acesso aleatório aos elementos da lista. Como já foi discutido, o acesso seqüencial possui realização mais eficiente.

```
--
-- Para manusear os pares de adjacentes, ao
-- invés de usar índices, usemos a função zip
-- aplicada a um par de listas construídas com base em xs
-- a)lista formada pelos elementos de xs, exceto o último,
--   que pode ser obtida com a função init;
-- b)lista formada pelos elementos de xs, exceto o primeiro,
--   que pode ser obtida com a função tail.
-- Com isso obtemos uma lista formada pelos pares adjacentes.
--
adjacentes xs = zip (init xs) (tail xs)

...

Main> adjacentes [1..10]
[(1,2),(2,3),(3,4),(4,5),(5,6),(6,7),(7,8),(8,9),(9,10)]
(414 reductions, 822 cells)
Main> adjacentes [10,9..1]
[(10,9),(9,8),(8,7),(7,6),(6,5),(5,4),(4,3),(3,2),(2,1)]
(364 reductions, 667 cells)
--
--  A nova função para definir lista dos maiores
--  que os sucessores. Agora trabalharemos com os pares
--
lms3 ps = [ (x,y) | (x,y) <- ps, x>y]
--
-- A nova versão de 'ord'
--
ord3 xs = lms3 (adjacentes xs) == []
```

```
...
Main> ord3 [1..10]
True
(313 reductions, 483 cells)
Main> ord3 [10,9..1]
False
(82 reductions, 145 cells)
```

O ganho obtido para a instância considerada foi de 86% no número de reduções e de 83% no número de células utilizadas, com respeito à segunda definição.

Exercícios:

1. Verificar se uma cadeia de caracteres representa um número inteiro positivo.
2. Dada uma lista l, contendo uma quantidade igual de números inteiros pares e ímpares (em qualquer ordem), defina uma função que, quando avaliada, produz uma lista na qual esses números pares e ímpares encontram-se alternados.
3. Dada uma lista xs, fornecer uma dupla contendo o menor e o maior elemento dessa lista.
4. Dadas duas listas de elementos distintos, determinar a união delas.

14. PARADIGMA APLICATIVO

14.1. INTRODUÇÃO: A descrição de funções, como acontece com outras formas de representação de conhecimento, admite vários estilos. Dependendo do problema que estamos tratando, um estilo pode ser mais conveniente que outros. Podemos dizer que influi muito na escolha, o quanto desejamos ou necessitamos, falar sobre como computar uma solução. A descrição de soluções tem um espectro de operacionalidade que vai do declarativo até o procedural, em outras palavras, do **que** desejamos computar ao **como** queremos computar.

Uma situação que ocorre com freqüência na descrição de funções é a necessidade de aplicar uma função, de forma cumulativa, à uma coleção de elementos. Em outras palavras, desejamos generalizar uma operação para que seu uso se estenda a todos os elementos de uma lista. Chamemos este estilo de paradigma aplicativo.

Por exemplo, suponha que desejamos obter a soma de todos os elementos de uma lista. A operação ***adição* (+)** segundo sabemos, é de natureza binária, ou seja, opera sobre dois elementos produzindo um terceiro. Para operar sobre todos os elementos de uma lista de forma a obter a soma total, podemos operá-los dois a dois, obtendo com isso resultados intermediários, que por sua vez poderão ser novamente operados e assim sucessivamente até que se obtenha o resultado final. Observemos o processo para uma instância:

Obter a soma dos elementos da lista [5, 9, 3, 8, 15, 16]

Podemos tomar os pares de elementos, primeiro com segundo, terceiro com quarto e assim sucessivamente.

expressão	nova expressão	redução
+ [5, 9, 3, 8, 15, 16]	+ [14, 3, 8, 15, 16]	5 + 9 = 14
+ [14, 3, 8, 15, 16]	+ [14, 11, 15, 16]	3 + 8 = 11
+ [14, 11, 15, 16]	+ [14, 11, 31]	15 + 16 = 31
+ [14, 11, 31]	+ [25, 31]	14 + 11 = 25
+ [25, 31]	+ [56]	25 + 31 = 56
+ [56]	56	

A escolha dos pares, no caso da adição, não importa. As reduções poderiam ser aplicadas em qualquer ordem, visto que a operação é adição é comutativa. Se, pode ser qualquer uma, então podemos estabelecer uma ordem, como por exemplo, da esquerda para a direita, usando em cada nova redução o elemento resultante da redução anterior.

Expressão	nova expressão	Redução
+ [5, 9, 3, 8, 15, 16]	+ [14, 3, 8, 15, 16]	5 + 9 = 14

+ [14, 3, 8, 15, 16]	+ [17, 8, 15, 16]	14 + 3 = 17
+ [17, 8, 15, 16]	+ [25, 15, 16]	17 + 8 = 25
+ [25, 15, 16]	+ [40, 16]	25 + 15 = 40
+ [40, 16]	+ [56]	40 + 16 = 56
+ [56]	56	

Em ***HUGS*** existe um operador que permite a descrição de computações desta natureza. Este operador denomina-se ***foldl***. A sua utilização requer ainda que seja indicado um ***valor especial*** para a operação considerada. Este elemento é tomado como ponto de partida para as reduções, ou seja, a primeira aplicação da operação é sobre o ***valor especial*** e o primeiro elemento da lista.

Eis a sintaxe:

foldl <operação> <valor especial> <lista>

A ordem estabelecida para as reduções é semelhante à ilustração acima, ou seja, caminha-se da esquerda para direita, usando o resultado da redução anterior para a nova redução.

expressão	nova expressão
foldl op ve [x0, x1, x2,...,xn]	***op*** [(op ve x0), x1, x2,...,xn]
op [(op (op ve x0) x1), x2,...,xn]	***op*** [op((op (op ve x0) x1) x2),...,xn]
op [op(op((op (op ve x0) x1) x2)... xn)]	op(op((op (op ve x0) x1) x2)... xn)

Merece aqui um destaque, foldl é na verdade um função, com 3 parametros, sendo que um deles é uma outra função. Dizemos neste caso que foldl é uma função de segunda ordem.

O tipo da operação tem que ser:

```
                <alfa> -> <beta> -> <beta>
onde :
   <alfa> pode ser do mesmo tipo de <beta>;
   <beta> tem que ser do tipo dos componentes
          da lista;
   o valor especial é do tipo <alfa>
```

Vejamos então a definição do operador ***sum*** disponível no HUGS e cujo objetivo é a descrição da soma dos elementos de uma lista.

```
sum xs = foldl (+) 0 xs
```

OBS:

1. Os operadores infixados são indicados entre parêntesis.
2. O valor especial é o zero, visto que não desejamos que o valor especial modifique o resultado da soma de todos os elementos. Entretanto, ele joga um papel muito especial quando a lista for vazia.

```
prelude> sum []
0
```

3. Em exemplos futuros veremos outros usos para o valor especial.

14.2. ALGUMAS OPERAÇÕES IMPORTANTES: Assim como a somatória, existem outras operações importantes, de grande utilidade, que podem ser obtidas pelas seguintes equações usando ***foldl.***

```
--
-- produto
--    valor especial = 1
--
product xs = foldl (*) 1 xs
--
-- conjunção
--    valor especial = True
--
and xs  = foldl (&&) True xs
--
-- disjunção
--    valor especial = False
--
or xs   = foldl (||) False xs
```

Exemplo 01: Usando "product" para descrever o fatorial de um número. Sabemos que: *O fatorial de um número natural n>0 é igual ao produto de todos os números naturais de 1 até n.*

```
--
-- definição de fatorial
--
fat n = product [1..n]
...

? fat 5
120

? fat 0
1

? fat 1
1
```

Exemplo 02: Podemos usar a disjunção (**or**) generalizada para definir uma função que avalia se em uma dada lista de números pelo menos um deles é impar.

```
--
-- pelo menos um impar
--
umImpar xs = or [odd x | x <- xs]
...

? umImpar [2,2,2,2,2,2,2,2]
False

? umImpar [2,2,2,1,2,2,2,2]
True

? umImpar []
False
```

14.3. O MENOR ELEMENTO DE UMA LISTA: Estamos interessados em obter uma função que associe uma lista xs com o elemento de xs que seja o menor de todos.

Anteriormente já apresentamos uma versão para a função que descreve o maior elemento de uma lista, que é bastante similar a esta. Na oportunidade exploramos uma propriedade que o elemento ***maior*** de todos deve satisfazer. No caso do menor elemento, podemos explorar uma propriedade análoga.

Em uma lista xs, dizemos que k é o menor elemento de xs, se e somente se a sublista de xs formada por elementos menores que k é vazia.

```
--
-- ´menores´ descreve os elementos menores que um
-- dado x em uma lista xs
--
menores x xs = [ y | y <- xs, y < x]
--  ´minimo´ descreve a sublista de xs dos elementos
--  que não possuem menores que eles em xs
--
minimos xs = [ k | k <- xs, menores k xs == [] ]
--
-- Como eles são todos idênticos podemos tomar o
-- primeiro deles
-- como solução de nosso problema.
--
menorL0 xs = head (menores xs)
```

Vamos explorar agora o problema a partir da generalização da operação **menor.** Em sua forma básica, a função **menor** associa dois números quaisquer com o menor entre eles. Precisamos identificar um elemento que não interfira no resultado para fazer o papel de ***valor especial***. Para a operação **menor** podemos observar que este papel pode ser desempenhado por qualquer um dos elementos da lista, visto que o menor entre dois valores idênticos é o próprio valor. Como pode ser qualquer um, podemos escolher o primeiro elemento de xs (head).

```
--
-- menor de dois
--
menor x y = if x < y then x else y
--
-- menor da lista
--
menorL xs = foldl menor (head xs)xs
...
? menorL [5,5,4,4,4,6,6,6,3,3,3,11,1,0]
0
(84 reductions, 157 cells)
? menorL [5]
5
(5 reductions, 13 cells)
? menorL [ ]

ERROR: Unresolved overloading
*** type        : Ord a => a
*** translation : menorL []
```

Podemos observar aqui que a função **menorL** é parcial pois não se aplica a lista vazia.

14.4. INSERÇÃO ORDENADA E ORDENAÇÃO DE UMA LISTA: A ordenação de dados é uma das operações mais realizadas em computação. Em todos os computadores do mundo, faz-se uso intensivo dela. Este assunto é muito especial e por isso mesmo profundamente estudado. Cabe-nos aqui fazer uma breve passagem pelo assunto, sem, contudo nos aprofundarmos nas questões de eficiência, que é central no seu estudo.

Será que podemos usar o conceito de generalização de uma operação para descrever a ordenação de uma lista? Que operação seria essa?

14.4.1. INSERÇÃO EM LISTA ORDENADA - Vamos começar discutindo outra questão mais simples: dada uma lista, com seus elementos já dispostos em ordem não decrescente, como descrever uma lista na mesma ordem, acrescida de um elemento também fornecido?

Podemos observar que se a lista xs está em ordem não decrescente, com respeito ao elemento x (dado), podemos descrevê-la através de dois segmentos:

xs = <menores que x> ++ <maiores ou iguais a x>

Para acrescentar x a xs, basta concatenar x entre os dois segmentos, obtendo a nova lista ys, assim:

ys = <menores que x em xs> ++ [x] ++ <maiores ou iguais a x em xs>

```
-- inserção ordenada
--
--
insord xs x = takeWhile (< x) xs
              ++ [x]
              ++ dropWhile (< x) xs
...

Main> insord [] 10
[10]
(27 reductions, 49 cells)
Main> insord [10] 20
[10,20]
(54 reductions, 84 cells)
Main> insord [10,20] 30
[10,20,30]
(79 reductions, 119 cells)
Main> insord [10,20,30] 5
[5,10,20,30]
(71 reductions, 110 cells)
Main> insord [5,10,20,30] 25
[5,10,20,25,30]
(126 reductions, 183 cells)
```

14.4.2. Ordenação - A APLICAÇÃO SUCESSIVA DE INSORD, CONFORME ILUSTRADA ACIMA, NOS DÁ UMA PISTA PARA NOSSA GENERALIZAÇÃO. PODEMOS PEGAR CADA ELEMENTO DA LISTA A SER ORDENADA E INSERI-LO EM ORDEM EM OUTRA LISTA QUE SERÁ PAULATINAMENTE CONSTRUÍDA, JÁ ORDENADA. VAMOS SEGUIR O EXEMPLO ACIMA, ONDE DESEJAMOS ORDENAR A LISTA [10,20,30,5,25]:

lista parcial	novo elemento	redução
[]	10	[10]
[10]	20	[10,20]
[10,20]	30	[10,20,30]
[10,20,30]	5	[5, 10, 20, 30]
[5, 10, 20, 30]	25	[5, 10, 20, 25, 30]

O ponto de partida, neste caso, é a lista vazia. Vamos tomar então a lista vazia como o valor especial. Vejamos como fica então nossa definição para ordenação de listas.

```
-- ordenação
--
ordena xs = foldl insord [] xs
...
Main> ordena [10,20,30,5,25]
[5,10,20,25,30]
(220 reductions, 344 cells)
Main> ordena [1..10]
[1,2,3,4,5,6,7,8,9,10]
(1055 reductions, 1505 cells)
Main> ordena [10,9..1]
[1,2,3,4,5,6,7,8,9,10]
(448 reductions, 712 cells)
```

14.5. INVERSÃO DE UMA LISTA: Dada uma lista xs, desejamos descrever a lista formada pelos elementos de xs tomados em ordem inversa.

Para resolver o problema precisamos inventar uma função básica passível de generalização.

Vamos começar descrevendo a função **insAntes.**

```
--
-- Insere um elemento antes do primeiro elemento
-- de uma dada lista. Função similar ao operador
-- (:) exceto pela ordem invertida dos parâmetros
--
insAntes xs x = x : xs

...
Main> insAntes [1..5] 0
[0,1,2,3,4,5]
(171 reductions, 255 cells)
Main> insAntes [] 0
[0]
(22 reductions, 29 cells)
```

Tentemos agora a generalização. A intenção é incluir cada elemento da lista xs que desejamos inverter, antes do primeiro elemento de uma lista que iremos construindo gradativamente. O ***valor especial*** será a lista vazia. Vejamos um exemplo onde inverteremos a lista [0, 1, 2, 3, 4]

lista parcial	novo elemento	redução
[]	0	[0]
[0]	1	[1, 0]
[1, 0]	2	[2, 1, 0]
[2, 1, 0]	3	[3, 2, 1, 0]
[3, 2, 1, 0]	4	[4, 3, 2, 1, 0]

Vamos então usar o operador foldl para construir a generalização desejada:

```
-- inversao
--
inverte xs = foldl insAntes [] xs

...

Main> inverte [0..4]
[4,3,2,1,0]
(172 reductions, 259 cells)
Main> inverte [4,3..4]
[4]
(74 reductions, 110 cells)
...
Main> reverse [4,3..4]
[4]
(74 reductions, 111 cells)
Main> reverse [0..4]
[4,3,2,1,0]
(171 reductions, 257 cells)
```

Observe a similaridade entre o desempenho da função **reverse**, pré-definida, com o desempenho da **inverte** que acabamos de definir.

14.6. INTERCALAÇÃO DE LISTAS: Dadas duas lista xs e ys, ambas em ordem não decrescente, desejamos descrever uma nova lista em ordem não decrescente, formada por todos os elementos das duas listas.

Um processo bastante conhecido, chamado de ***balance line***, consiste em ir transferindo para a nova lista, os elementos de uma das listas de entrada, enquanto estes forem menores que os da outra. Quando a condição não é mais satisfeita, fazemos o mesmo para a outra lista.

Por exemplo, vamos intercalar as listas [2, 4, 6, 8] e [1, 3, 5]. Vejamos o desenrolar do processo.

lista parcial	Andamento em xs	Andamento em ys	Redução
[]	[2, 4, 6, 8]	[1, 3, 5]	[1]
[1]	[2, 4, 6, 8]	[3, 5]	[1, 2]
[1, 2]	[4, 6, 8]	[3, 5]	[1, 2, 3]
[1, 2, 3]	[4, 6, 8]	[5]	[1, 2, 3, 4]
[1, 2, 3, 4]	[6, 8]	[5]	[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]	[6, 8]	[]	[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6]	[8]	[]	[1, 2, 3, 4, 5, 6, 8]
[1, 2, 3, 4, 5, 6, 8]	[]	[]	

Vamos precisar aqui de uma bela engenharia para a construção da função a generalizar. Desta vez, nossa entrada é formada por duas listas e como sabemos o

operador **foldl**, a cada vez, processa um elemento de uma determinada lista. Isto nos leva a ter que inventar também a lista a ser processada.

A função básica pode ser inventada a partir da descrição de um passo do processo que denominamos de ***balance line***. Em cada passo temos como entrada uma lista parcialmente construída e duas listas parcialmente processadas. Como saída teremos mais um passo de construção da lista resultante e o andamento no processamento de uma das listas. Na lista em construção devemos inserir o menor entre os cabeças (head) das duas listas. Aquela que tiver o menor cabeça dará origem a uma nova lista, da qual foi excluído o primeiro elemento, ou seja, será idêntica ao resto (tail) da lista escolhida.

```
--
-- Descrição de um passo do Balance Line.
-- 1) Quando uma das duas listas for vazia a
--    outras é diretamente concatenada no final
--    da lista em construção.
--
passoBL (xs,ys,zs) = if (ys == [])
                     then ([],[],zs++xs)
                     else if (xs == [])
                          then ([],[],zs++ys)
                          else if (head xs <= head ys)
                                    then (tail xs,ys,zs++[head xs])
                               else (xs, tail ys,zs++[head ys])
...
... aplicação sucessiva da função passoBL à intercalação
... das listas acima propostas
...
Main> passoBL ([2,4,6,8],[1,3,5],[])
([2,4,6,8],[3,5],[1])
(121 reductions, 219 cells)
Main> passoBL ([2,4,6,8],[3,5],[1])
([4,6,8],[3,5],[1,2])
(122 reductions, 222 cells)
Main> passoBL ([4,6,8],[3,5],[1,2])
([4,6,8],[5],[1,2,3])
(123 reductions, 225 cells)
Main> passoBL ([4,6,8],[5],[1,2,3])
([6,8],[5],[1,2,3,4])
(124 reductions, 228 cells)
Main> passoBL ([6,8],[5],[1,2,3,4])
([6,8],[],[1,2,3,4,5])
(128 reductions, 239 cells)
Main> passoBL ([6,8],[],[1,2,3,4,5])
([],[],[1,2,3,4,5,6,8])
(116 reductions, 223 cells)
...
Main> passoBL ([1..5],[],[])
([],[],[1,2,3,4,5])
(190 reductions, 329 cells)
Main> passoBL ([],[1..5],[])
([],[],[1,2,3,4,5])
(193 reductions, 339 cells)
Main> passoBL ([1..3],[4..5],[])
([2,3],[4,5],[1])
(219 reductions, 371 cells)
```

A generalização, de forma que possamos aplicar a função gradativamente, de tal forma que todo o processo se complete, requer a invenção de uma lista sobre a qual ocorra a repetição. O número de aplicações sucessivas será no máximo igual à soma dos comprimentos das listas a serem intercaladas. Podemos então definir a aplicação sobre a lista dos números de 1 até a soma dos tamanhos.

```
  --
-- Função para Intercalar duas listas ordenadas
-- 1) a função passoBL é repetidamente aplicada, sobre
--    o resultado obtido no passo anterior;
-- 2) a função passoBL é binária, como exigido por foldl;
-- 3) a aplicação sucessiva é controlada pelo comprimento
--    da lista resultante;
-- 4) o resultado final é obtido pela seleção do terceiro
--    elemento da tripla através da primitiva "thd3"
--
baLine xs ys  = thrd3 (baLine3 xs ys)
--
thrd3 (x,y,z) = z
--
baLine3 xs ys = foldl passoBL (xs,ys,[]) [1..tr]
                where
                   tr = (length xs)+(length ys)
--
-- Descrição de um passo do Balance Line.
-- 1) Quando uma das duas listas for vazia a
--    outras é diretamente concatenada no final
--    da lista em construção.
-- 2) O parâmetro k é apenas para estabelecer o tipo
--    binário exigido por foldl
--
passoBL (xs,ys,zs) k = if (ys == [])
                       then ([],[],zs++xs)
                       else if (xs == [])
                            then ([],[],zs++ys)
                            else if (head xs <= head ys)
                                 then (tail xs,ys,zs++[head xs])
                                 else (xs, tail ys,zs++[head ys])

...

Main> thrd3 (10,20,30)
30
(11 reductions, 12 cells)
Main> baLine3 [1..3] [4..6]
([],[],[1,2,3,4,5,6])
(514 reductions, 896 cells)
Main> baLine [1..3] [4..6]
[1,2,3,4,5,6]
(485 reductions, 784 cells)
Main> baLine [1..5] [3..7]
[1,2,3,3,4,4,5,5,6,7]
(807 reductions, 1326 cells)
Main> baLine [2, 4, 6, 8]  [1, 3, 5]
[1,2,3,4,5,6,8]
(425 reductions, 696 cells)
```

Exercícios: Use o paradigma aplicativo para resolver os problemas abaixo:

1. Dadas duas strings xs e ys, verificar se xs é sublista de ys.
2. Definir a função tWhile, que tenha o mesmo comportamento que a função takeWhile.
3. Definir a função dWhile, que tenha o mesmo comportamento que a função dropWhile.
4. Verificar se uma string é um palíndrome (a string é a mesma quando lida da esquerda para a direita ou da direita para a esquerda).

15. Processamento de Cadeias de Caracteres – primeiros passos

15.1. INTRODUÇÃO: Além de números, nosso mundo é povoado por textos. Cada vez mais se torna presente o uso de computadores para nos auxiliar na tarefa de armazenar, recuperar e processar documentos. Neste capítulo estamos interessados em fazer uma breve introdução ao uso de computadores nessas tarefas.

O ponto de partida é o tipo caracter (char), que nos permite representar textos na memória (principal e secundária) dos computadores. Veremos também como agrupá-los para compor palavras, frases e por fim documentos.

15.2. O TIPO CHAR: O tipo char é formado por um conjunto de símbolos. Outro nome usado para esta coleção é alfabeto, ou seja, o conjunto de átomos que servirão de base para a construção de cadeias complexas, inclusive os textos usuais. Entre os símbolos citados podemos destacar três agrupamentos relevantes, tais como:

1. As letras maiúsculas do alfabeto;
2. As letras minúsculas do alfabeto;
3. Os algarismos arábicos;

Estes três agrupamentos gozam de uma propriedade muito importante. Dentro deles os símbolos possuem uma relação de ordem, de tal forma que podemos usar a noção usual de ordenação para letras e algarismos.

Além destes, podemos citar ainda os sinais de pontuação e alguns símbolos com funções especiais, como, por exemplo, indicador de final de linha de texto. Os símbolos são sempre apresentados em HUGS entre aspas simples, para que possam ser diferenciados dos nomes de parâmetros, funções e outros. Por exemplo, a letra ***a*** deve ser denotada por ***'a' .***

A definição a seguir ajuda a esclarecer esta necessidade.

$$f\ a = (a, 'a')$$

Aqui usamos a letra “a” três vezes. A primeira, da esquerda para direita, nomeia um parâmetro da função “f”. Na segunda ocorrência, utilizamos o parâmetro da função para se constituir do primeiro elemento de uma tupla. Já a terceira ocorrência se refere à constante “a”.

Vejamos as respostas do sistema para diferentes usos da função **f**

```
1 Main> f 3
  (3,'a')
2 Main> f 5
  (5,'a')
3 Main> f a
  ERROR - Undefined variable "a"
4 Main> f 'a'
  ('a','a')
5 Main> f 'aa'
  ERROR - Improperly terminated character constant
6 Main> f 'b'
  ('b','a')
7 Main> f a where a='a'
('a','a')
```

Nas situações 1, 2, e 6, o parâmetro **a** é instanciado para os valores 3, 5 e ‘b’ , resultando nos pares (3, ‘a’), (5,’a’) e (´b, ‘a’). Na situação 4, o parâmetro **a** é instanciado para a constante ‘a’, produzindo o par (‘a’, ‘a’). Na situação 3, o uso de a está incorreto, pois como não está entre as aspas simples, é interpretado como um parâmetro, que não está instanciado quando deveria estar. Na situação 7, ao contrário da situação 3, o valor de **a** é instanciado através da cláusula **where** e a situação fica similar à situação 4.

A coleção total dos símbolos de nosso alfabeto forma uma seqüência de tal forma que podemos fazer o mapeamento entre a subseqüência de números naturais, de zero (0) a 255 e a seqüência de símbolos.

Duas funções básicas permitem que se faça a conversão entre as duas seqüências. Elas podem ser encontradas em uma biblioteca de funções diferente de Prelude.hs. Essa biblioteca se chama Data.Char. Para usar essas funções no seu interpretador HUGS, é preciso executar o comando **:l Data.Char** no interpretador:

1. A função ***chr*** associa um número natural no intervalo [0,255] com o caracter correspondente. Por exemplo, ***chr*** 64 = '@' e ***chr*** 38 = '&'.

2. A função ***ord*** faz o mapeamento inverso, ou seja, associa um símbolo com o número natural correspondente. Por exemplo, ***ord*** '?' = 63 e ***ord*** '%' = 37.

A tabela a seguir apresenta alguns dos agrupamentos importantes.

		Algarismos		Letras Maiúsculas				Letras Minúsculas			
32	' '	48	**0**	65	**A**	91	**[**	97	**a**	123	**{**
33	**!**	49	**1**	66	**B**	92	\	98	**b**	124	|
34	**"**	50	**2**	67	**C**	93	**]**	99	**c**	125	**}**
35	**#**	51	**3**	68	**D**	94	^	100	**d**	126	~
36	**$**	52	**4**	69	**E**	95	_	101	**e**		
37	**%**	53	**5**	70	**F**	96	`	102	**f**		
38	**&**	54	**6**	71	**G**			103	**g**		
39	\	55	**7**	72	**H**			102	**h**		
40	**(**	56	**8**	73	**I**			105	**i**		
41	**)**	57	**9**	74	**J**			106	**j**		
42	*	58	**:**	75	**K**			107	**k**		
43	+	59	**;**	76	**L**			108	**l**		
44	**,**	60	<	77	**M**			109	**m**		
45	**-**	61	=	78	**N**			110	**n**		
46	**.**	62	>	79	**O**			111	**o**		
47	/	63	**?**	80	**P**			112	**p**		
		64	**@**	81	**Q**			113	**q**		
				82	**R**			114	**r**		
				83	**S**			115	**s**		
				84	**T**			116	**t**		
				85	**U**			117	**u**		
				86	**V**			118	**v**		
				87	**W**			119	**w**		
				88	**X**			120	**x**		
				89	**Y**			121	**y**		
				90	**Z**			122	**z**		

Além das operações **ord** e **chr**, podemos contar com os operadores relacionais que foram apresentados no Capítulo 6. Da mesma forma que existem os operadores para

comparação de números, existe os operadores relacionais que comparam dois símbolos, como no exemplo:

```
Prelude> 'c' > 'h'
False
```

O significado de uma operação relacional sobre characteres é determinado com base na ordem destes com respeito à tabela apresentada para codificação desses símbolos. Assim, podemos dizer que:

Se x e y são do tipo caractere, **x > y** se e somente se **ord x > ord y**

Podemos agora construir algumas definições interessantes, conforme se apresenta nos quadros a seguir:

```
--
-- verifica se um dado símbolo é letra
--
letra x = (maiuscula x) || (minuscula x)
--
maiuscula      x = pertence x ('A','Z')
minuscula      x = pertence x ('a','z')
pertence x (a,b) = (x >= a) && (x <= b)
```

```
--
-- verifica se um símbolo é um algarismo
--
algarismo      x = pertence x ('0','9')
--
-- Associa uma letra minuscula com a sua
-- correspondente maiuscula e mantém o
-- símbolo fornecido nos demais casos
--
caps x = if minuscula x
         then chr (ord x - 32)
         else x
```

```
--
-- Determina a posição relativa de uma letra
-- dentro do alfabeto, onde as maiúsculas e
-- minúsculas possuem a mesma posição.
-- Outros símbolos devem ser associados ao
-- valor 0 (zero)
--
ordAlfa x = if letra x
            then ord (caps x) - 64
            else 0
```

Eis mais alguns exemplos de funções com o tipo char.

```
--
-- verifica se um símbolo é uma letra vogal
--
vogal x      = letra x && ocorre (caps x) vogais
               where
                 vogais = ['A','E','I','O','U']
--
ocorre x xs = or [x == y | y<- xs]
```

```
--
-- verifica se um símbolo é uma letra
-- consoante
--
consoante x = letra x && (not (vogal x))
--
```

```
--
-- Descreve uma lista de pares onde o primeiro termo
-- é um número entre 0 e 255 e o segundo o caracter
-- correspondente. O intervalo desejado é informado
-- por um par de valores no intervalo 0 a 255.
--
tabOrdChr (i,f) = [(x, chr x) | x <- [i..f]]
```

Vejamos alguns exemplos de uso das definições acima apresentadas:

```
Main> tabOrdChr (65,70)
[(65,'A'),(66,'B'),(67,'C'),(68,'D'),(69,'E'),(70,'F')]
(357 reductions, 604 cells)
Main> tabOrdChr (97,102)
[(97,'a'),(98,'b'),(99,'c'),(100,'d'),(101,'e'),(102,'f')]
(357 reductions, 607 cells)
Main> tabOrdChr (45,50)
[(45,'-'),(46,'.'),(47,'/'),(48,'0'),(49,'1'),(50,'2')]
(357 reductions, 604 cells)
```

15.3. O TIPO STRING: Podemos agrupar átomos do tipo caracter (char) para formar o que denominamos de cadeia de caracteres, usualmente chamadas de "palavra". O mecanismo para agregar é o construtor da lista. Podemos por exemplo, escrever em HUGS a lista ['c','i','d','a','n','i','a'] para representar a palavra ***cidadania.***

Assim procedendo podemos escrever qualquer texto, uma vez que um texto não é nada mais nada menos que uma longa cadeia de símbolos envolvendo letras e sinais de pontuação. A separação de palavras é obtida pelo uso de um caracter especial, denominado de ***espaço***, que tem representação interna igual a 32. Por exemplo, para representar a expressão "Vamos nessa!", usamos a lista:

['V','a','m','o','s',' ','n','e','s','s','a','!'].

Uma boa notícia é que o HUGS nos proporciona uma maneira mais amistosa para tratar com lista de caracteres. Em HUGS podemos representar uma lista de caracteres, envolvendo a cadeia por aspas duplas, o que por certo, além de mais elegante e legível, no poupa trabalho. Por exemplo, a cadeia acima representada por ser também escrita na seguinte forma:

"Vamos nessa!"

À uma lista de caracteres o HUGS associa um tipo sinônimo, denominado ***string***. Assim, as duas representações acima são idênticas, conforme podemos observar na avaliação abaixo:

```
Prelude> "Vamos nessa!" == ['V','a','m','o','s',' ','n','e','s','s','a','!']
True
(64 reductions, 102 cells)
```

Um bom lembrete é que, cadeias de caracteres são definidas como listas, e como tais, podemos usar sobre elas todas as operações que sabemos até agora sobre listas.

Por exemplo, podemos construir uma função para contar a quantidade de vogais existente em uma cadeia.

```
contaVogais xs = length vogais
                 where
                    vogais = [x | x <-xs, vogal x]
...
Main> contaVogais "Vamos nessa!"
4
(966 reductions, 1383 cells)
```

15.4. FUNÇÕES BÁSICAS PARA O TIPO STRING: Uma cadeia de caracteres, por ser uma lista, herda todas as operações que já apresentamos para as listas. Além dessas, podemos contar ainda com algumas operações que apresentamos a seguir:

words xs - Associa uma cadeia de caracteres com a lista de *palavras* nela contida. Entende-se por *palavra* um agrupamento de símbolos diferentes do símbolo ***espaço (" ").***

```
Prelude> words "Se temos que aprender a fazer, vamos
aprender fazendo!"
["Se","temos","que","aprender","a","fazer,",
"vamos","aprender","fazendo!"]
(2013 reductions, 3038 cells)
```

Podemos observar que os símbolos usuais de separação (por exemplo, "," e "!" são considerados como parte das *palavras*. Vamos agora construir uma função que considere os símbolos usuais de separação, além do espaço.

```
--
-- A função palavras
--
palavras xs = [ takeWhile letra x | x <- words xs]
--
--
--
...

Main> palavras "Se temos que aprender a fazer, vamos aprender fazendo!"
["Se","temos","que","aprender","a","fazer","vamos","aprender","fazendo"]
(3869 reductions, 5444 cells)


Main> palavras "jose123 maria456 joana!!!"
["jose","maria","joana"]
(1445 reductions, 2081 cells)


Main> words "jose123 maria456 joana!!!"
["jose123","maria456","joana!!!"]
(951 reductions, 1423 cells)
```

De fato, a função **palavras** trata a questão proposta: abandonar os símbolos de pontuação. Acontece que ela abandona muito mais que isso, como podemos ver no exemplo, onde a cadeia "jose123" perde o seu sufixo numérico, tornando-se apenas "jose".

Vamos construir uma nova função então onde isso possa ser resgatado.

```
--
-- A função palavras1
--
palavras1 xs = [ takeWhile alfa x | x <- words xs]
      where alfa x = letra x || algarismo x
--
--
--
...

Main> palavras1  "x123 y456 aux!!!"
["x123","y456","aux"]
```

Bom, parece que agora temos uma solução adequada.

Exercícios

1. Verificar se uma letra é vogal.
2. Verificar se uma letra é consoante.
3. Gerar a lista de pares onde o primeiro termo é um número entre 0 e 255 e o segundo termo é o caracter correspondente pela tabela ASCII. O intervalo de valores desejados é informado por um par de valores entre 0 e 255.
4. Dadas duas strings xs e ys, verificar se xs é prefixo de ys.
5. Dadas duas strings xs e ys, verificar se xs é sufixo de ys.
6. Dadas duas strings xs e ys, verificar se xs é sublista de ys.

16. O PARADIGMA RECURSIVO

16.1. INTRODUÇÃO: Como já falamos anteriormente, existem várias maneiras de definir um conceito. A essas maneiras convencionamos chamar de paradigmas. Aqui trataremos de mais um destes, o paradigma recursivo. Dizer que trataremos de mais um é simplificar as coisas, na verdade este paradigma é um dos mais ricos e importantes para a descrição de computações. O domínio deste paradigma é de fundamental importância para todo aquele que deseja ser um *expert* em Programação de Computadores enquanto ciência e tecnologia.

De uma maneira simplificada podemos dizer que o núcleo deste paradigma consiste em descrever um conceito de forma recursiva. Isto equivale a dizer que definiremos um conceito usando o próprio conceito. Apesar de disto parecer muito intrigante, não se assuste, aos poucos, quando esboçarmos melhor a idéia ela se mostrará precisa, simples e poderosa.

Vamos pensar num conceito bem corriqueiro. Que tal definirmos o conceito ***escada***.

Como podemos descrever escada usando a própria escada? A resposta é bem simples:

Uma escada é igual a um degrau seguido de uma escada **(Figura 15.1).**

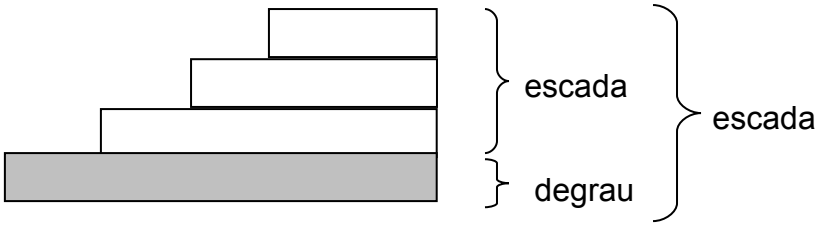


Figura 15.1 – Uma escada

Fácil não é? Será que isto basta? Onde está o truque? Parece que estamos andando em círculo, não é mesmo? Para entender melhor vamos discutir a seguir alguns elementos necessários para a utilização correta da recursão na definição de novas funções.

16.2. DESCRIÇÃO RECURSIVA DE UM CONCEITO FAMILIAR: Antes de avançar em nossa discussão vamos apresentar mais um exemplo. Desta vez usaremos um que é bastante familiar para alunos de ciências exatas. Estamos falando da descrição do fatorial de um número. Já vimos neste curso uma forma de descrever este conceito dentro do HUGS quando estudamos o paradigma aplicativo. Na oportunidade usamos a seguinte descrição:

*O fatorial de um número natural **n** > 0 é igual*
*ao produto de todos os números naturais de 1 até **n**.*

Ou ainda em notação mais formal:

$$n ! = 1 \times 2 \times 3 \times \ldots \times n$$

Em HUGS, como já vimos, teremos a seguinte definição:

```
--
-- definição (aplicativa) de fatorial
--
fat n = product [1..n]
...

? fat 5
120
? fat 0
1
? fat 1
1
```

Há uma outra forma de definir, também familiar aos alunos do primeiro ano universitário:

O Fatorial de um número natural n é igual ao produto deste número pelo fatorial de seu antecessor.

Novamente, sendo mais formal, podemos escrever:

$$n ! = n \times (n - 1) !$$

E em HUGS, como ficaria? Que tal a definição a seguir?

```
--
-- definição recursiva de fatorial
--
fat n = n * fat (n - 1)
```

Vamos exercitar a definição:

```
Main> fat 5

(23967 reductions, 47955 cells)
ERROR: Control stack overflow
```

Bom, parece que houve um pequeno problema com nossa definição. A avaliação de **fat 5** produziu uma situação de erro. Vamos deixar para entender este erro melhor para depois. Por enquanto já podemos adiantar que ele foi provocado por um pequeno esquecimento de nossa parte.

Na verdade a nossa definição recursiva para fatorial estava incompleta. Esta que exibimos só se aplica aos naturais maiores que zero. A definição do fatorial de zero não é recursiva, ela é independente:

O fatorial de zero é igual a 1.

Temos então duas definições para fatorial e precisamos integrá-las. Vejamos uma tentativa:

O Fatorial de um número natural n é:

1. *igual a 1 se n=0;*
2. *igual ao produto deste número pelo fatorial de seu antecessor, se n > 0*

Vamos ver como essa integração pode ser feita em HUGS. Podemos de imediato observar que trata-se de uma definição condicional e logo nos vem a lembrança de que nossa linguagem possui um mecanismo, as expressões condicionais.

```
--
-- definição recursiva de fatorial
-- (corrigida)
--
fat n = if n==0
        then 1
        else n * fat (n - 1)
```

Vamos submeter algumas situações para o HUGS:

```
Main> fat 5
120
(79 reductions, 124 cells)
Main> fat 20
2432902008176640000
(258 reductions, 466 cells)
Main> fat 0
1
(18 reductions, 18 cells)
```

Pelo visto agora deu tudo certo.

16.3. ELEMENTOS DE UMA DESCRIÇÃO RECURSIVA: Em uma descrição recursiva devemos ter em conta certo elementos importantes. É fundamental que todos eles sejam contemplados para que nossas descrições estejam corretas. O exemplo anteriormente apresentado é suficiente para ilustrar todos eles. Vamos então discuti-los:

Definição geral : Toda definição recursiva tem duas partes, uma delas se aplica a um valor qualquer do domínio do problema, denominamos de geral. Esta tem uma característica muito importante, o conceito que está sendo definido deve ser utilizado. Por exemplo, para definir fatorial de n, usamos o fatorial do *antecessor de n*. Observe aqui, entretanto que o mesmo conceito foi utilizado, mas não para o mesmo valor. Aplicamos o conceito a um valor mais simples, neste caso o antecessor de n.

Definição independente : A outra parte da definição é destinada ao tratamento de um valor tão simples que a sua definição possa ser dada de forma independente. Este elemento é também conhecido como base da recursão. No caso do fatorial, o valor considerado é o zero.

Obtenção de valores mais simples : Para aplicar o conceito a um valor mais simples precisamos de uma função que faça este papel. No caso do fatorial, usamos a subtração de n por 1, obtendo assim o antecessor de n. Em cada caso, dependendo do domínio do problema e do problema em si, precisaremos encontrar a função apropriada.

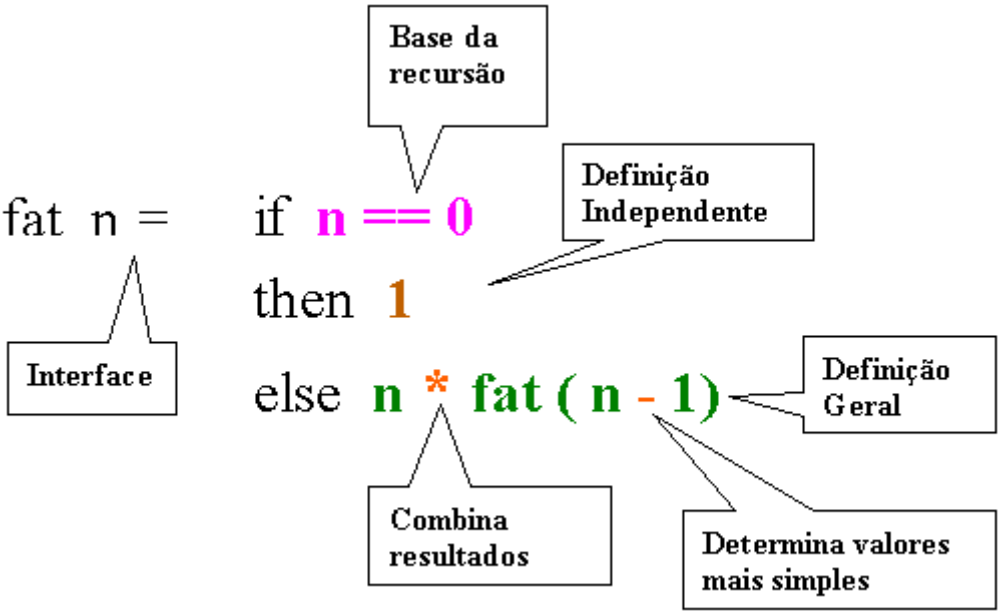
Função auxiliar : Na definição geral, para obter um valor usando o valor considerado e o valor definido recursivamente, em geral faz-se necessário o uso de uma função auxiliar. Algumas vezes esta função pode ser originada a partir de um conceito aplicável a dois elementos e que desejamos estender aos elementos de uma lista. Um exemplo é o caso da somatória dos elementos de uma lista, como veremos adiante. No caso do fatorial esta função é a multiplicação.

Garantia de atingir o valor independente : É fundamental que a aplicação sucessiva da função que obtém valores mais simples garanta a determinação do valor mais simples. Este valor é também denominado de base da recursão. Por exemplo, no caso do fatorial, sabemos que aplicando a subtração sucessivas vezes produziremos a seqüência:

n, (n-1), (n-2), ... 0

Esta condição é fundamental para garantir que ao avaliarmos uma expressão atingiremos a base da recursão.

Voltemos à definição do fatorial para destacarmos os elementos acima citados, como podemos observar no quadro esquemático a seguir:



16.4. AVALIANDO EXPRESSÕES: A esta altura dos acontecimentos a curiosidade sobre como avaliar expressões usando conceitos definidos recursivamente já deve estar bastante aguçada. Não vamos, portanto retardar mais essa discussão. Apresentamos a seguir um modelo bastante simples para que possamos entender como avaliar expressões que usam conceitos definidos recursivamente. Novamente não precisaremos entender do funcionamento interno de um computador nem da maneira como uma determinada implementação de HUGS foi realizada. Basta-nos o conceito de redução que já apresentamos anteriormente.

Relembremos o conceito de redução. O avaliador deve realizar uma seqüência de passos substituindo uma expressão por sua definição, até que se atinja as definições primitivas e os valores possam ser computados diretamente.

Vamos aplicar então este processo para realizar a avaliação da expressão ***fat 5***

passo	Redução	Justificativa
0	**fat 5**	expressão proposta
1	**5 * fat 4**	substituindo fat por sua definição geral
2	**5*(4 * fat 3)**	Idem
3	**5*(4* (3 * fat 2)**)	Idem
4	**5*(4*(3*(2 * fat 1))**)	Idem
5	**5*(4*(3*(2*(1 * fat 0)**	Idem
6	**5*(4*(3*(2*(1 * 1))))**	usando a definição específica
7	**5*(4*(3*(2*1)))**	usando a primitiva de multiplicação
8	**5*(4*(3*2)**	Idem
9	**5*(4*6)**	Idem
10	**5 * 24**	Idem
11	**120**	Idem

Surpreso(a)? Simples, não? É assim mesmo, bem simples. A cada passo vamos substituindo uma expressão por outra até que nada mais possa ser substituído. O resultado surgirá naturalmente. Mais tarde voltaremos ao assunto.

16.5. RECURSÃO EM LISTAS: A esta altura deste curso já estamos certos que o uso de lista é indispensável para escrever programas interessantes. Em vista disso, nada mais óbvio que perguntar sobre o uso de recursão em listas. Veremos que o uso de definições recursivas em listas produz descrições simples, precisas e elegantes.

Já está na hora de alertar que os valores sobre os quais aplicamos os conceitos que queremos definir recursivamente possuem uma característica importantíssima, eles em si são recursivos.

Por exemplo, qualquer valor pertencente aos naturais pode ser descrito a partir da existência do zero e da função sucessor (suc). Vejamos como podemos obter o valor 5:

5 = suc(suc(suc(suc(suc 0)))))

As listas são valores recursivos. Podemos descrever uma lista da seguinte maneira:

Uma lista é:

1. a lista vazia;
2. um elemento seguido de uma lista

Esta natureza recursiva das listas nos oferece uma oportunidade para, com certa facilidade, escrever definições recursivas. A técnica consiste basicamente em:

1. Obter a definição geral: isto consiste em identificar uma operação binária simples que possa ser aplicada a dois valores. O primeiro deles é o **primeiro** (**head**) da lista e o outro é um valor obtido pela aplicação do conceito em definição ao resto (**tail**) da lista;
2. Obter a definição independente, que se aplicará à base da recursão. Esta, em geral, é a ***lista vazia***;
3. Garantir que a aplicação sucessiva do **tail** levará à base da recursão.

Na Figura 15.2 ilustramos o processo recursivo de obter listas cada vez menores, através da aplicação da função **tail**. Ao final do processo obteremos a lista vazia ([ ]).

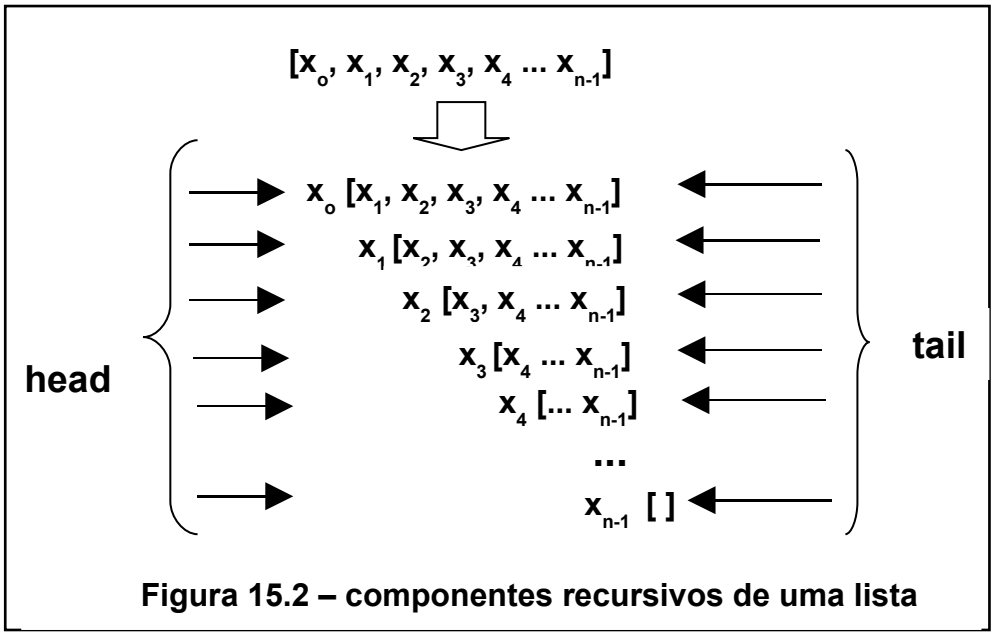


Figura 15.2 – componentes recursivos de uma lista

Exemplo 01 - Descrever o somatório dos elementos de uma lista.

Solução - Podemos pensar da seguinte maneira: o somatório dos elementos de uma lista é igual à soma do primeiro elemento da lista como o somatório do resto da lista. Além disso, o somatório dos elementos de uma lista vazia é igual a zero.

$$\text{somat } [x_0, x_1, x_2, x_3, x_4 \ldots x_{n-1}]$$
$$x_0 + \text{somat } [x_1, x_2, x_3, x_4 \ldots x_{n-1}]$$
$$X_0 + x_1 + \text{somat } [x_2, x_3, x_4 \ldots x_{n-1}]$$
$$X_0 + x_1 + x_2 + \text{somat } [x_3, x_4 \ldots x_{n-1}]$$
$$X_0 + x_1 + x_2 + x_3 + \text{somat } [x_4 \ldots x_{n-1}]$$
$$X_0 + x_1 + x_2 + x_3 + x_4 + \text{somat } [\ldots x_{n-1}]$$
$$X_0 + x_1 + x_2 + x_3 + x_4 + \ldots + x_{n-1} + \text{somat } [\,]$$
$$X_0 + x_1 + x_2 + x_3 + x_4 + \ldots + x_{n-1} + 0$$

Figura 15.2 – desenvolvimento da computação de somatório dos elementos de uma lista

Vejamos então a codificação:

```
--
-- definição recursiva da somatória dos
-- elementos de uma lista
--
somat xs = if null xs
           then 0
           else head xs + somat (tail xs)
```

A seguir alguns exemplos de uso:

```
Main> somat [4,5,2,7,9]
27
(52 reductions, 60 cells)
Main> somat [1..10]
55
(275 reductions, 421 cells)
Main> somat [1000,9999..1]
0
(45 reductions, 68 cells)
Main> somat [1000,999..1]
500500
(18051 reductions, 25121 cells)
```

Exemplo 02 - Descrever a função que determina o elemento de valor máximo uma lista de números.

Solução: O máximo de uma lista é o maior entre o primeiro elemento da lista e o máximo aplicado ao resto da lista. Uma lista que tem apenas um elemento tem como valor máximo o próprio elemento (Figura 15.3).

maximo $[x_o, x_1, x_2, x_3, x_4 \ldots x_{n-1}]$
maior(x_o , maximo $[x_1, x_2, x_3, x_4 \ldots x_{n-1}]$)
maior(x_o , maior(x_1, maximo $[x_2, x_3, x_4 \ldots x_{n-1}]$))))
maior(x_o , maior(x_1, maior(x_2, maximo $[x_3, x_4 \ldots x_{n-1}]$)))
maior(x_o , maior(x_1, maior(x_2, maior(x_3, maximo $[x_4 \ldots x_{n-1}]$))))
maior(x_o , maior(x_1, maior(x_2, maior(x_3, maior(x_4, ... maximo$[x_{n-2}, x_{n-1}]$))))))
maior(x_o , maior(x_1, maior(x_2, maior(x_3, maior(x_4, ... maior(x_{n-2}, maximo[x_{n-1}])))))

Figura 15.3 – desenvolvimento da computação do elemento máximo de uma lista

A definição recursiva é apresentada a seguir:

```
--
-- definição recursiva do máximo de uma lista
--
maximo xs = if null (tail xs)
            then head xs
            else maior (head xs) (maximo (tail xs))
--
maior x y = if x > y then x else y
```

E vamos acompanhar agora algumas submissões:

```
Main> maximo [4,6,7,89,32,45,98,65,31]
98
(126 reductions, 150 cells)
Main> maximo ([1..1000]++[1500,1400..1])
```

```
1500
(31419 reductions, 44567 cells)
Main> maximo [100,(100 - k) .. 1] where k = 80
100
(91 reductions, 125 cells)
```

E agora uma surpresa.

```
Main> maximo [ ]
ERROR: Unresolved overloading
*** Type       : Ord a => a
*** Expression : maximo []
```

Você consegue explicar?

Exemplo 03 - Descrever a função que verifica se um dado valor ocorre em uma lista também dada.

Solução : Podemos pensar da seguinte maneira: Um dado elemento k ocorre em uma lista se ele é igual ao primeiro elemento da lista ou se ele ocorre no resto da lista. Em uma lista vazia não ocorrem elementos quaisquer (Figura 15.4).

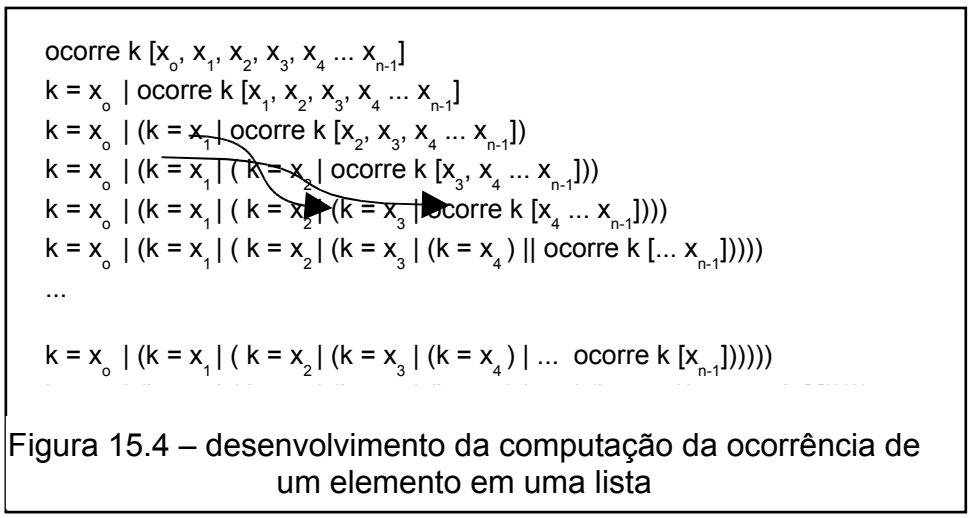


Figura 15.4 – desenvolvimento da computação da ocorrência de um elemento em uma lista

Vejamos então a codificação:

```
--
-- descreve a ocorrência de dado k em uma lista xs
--
ocorre k xs = if null xs
              then False
              else (k==head(xs)) || ocorre k (tail xs)
```

E algumas submissões:

```
Main> ocorre 5 [8,65,46,23,99,35]
False
(71 reductions, 111 cells)
Main> ocorre 5 [8,65,46,5,23,99,35]
True
(47 reductions, 58 cells)
Main> ocorre 5 [ ]
False
(16 reductions, 30 cells)
```

Exemplo 04 - Descrever a função que obtém de uma lista xs a sublista formada pelos elementos que são menores que um dado k :

Solução : Precisamos descrever uma nova lista, vamos denominá-la de ***menores***, em função de xs e de k. Quem será esta nova lista? Se o primeiro elemento de xs for menor que k, então ele participará da nova lista, que pode ser descrita como sendo formada pelo primeiro elemento de xs seguido dos menores que k no resto de xs. Se por outro lado o primeiro não é menor que k, podemos dizer que a lista resultante é obtida pela aplicação de menores ao resto da lista. Novamente a base da recursão é definida pela lista vazia, visto que em uma lista vazia não ocorrem elementos menores que qualquer k.

A codificação é apresentada a seguir:

```
--
-- define a lista de menores que um dado elemento em
-- uma lista dada
--
menores k xs = if null xs
               then xs
               else if head xs < k
                       then head xs : menores k (tail xs)
                       else menores k (tail xs)
```

Desta podemos obter as seguintes avaliações:

```
Main> menores 23 [8,65,46,5,23,99,35]
[8,5]
(122 reductions, 188 cells)
Main> menores 46 [8,65,46,5,23,99,35]
[8,5,23,35]
(135 reductions, 175 cells)
Main> menores 5 []
[]
(17 reductions, 24 cells)
```

16.6. EXPLORANDO REUSO: Segundo o Professor George Polya, após concluir a solução de um problema, devemos levantar questionamentos a respeito das possibilidades de generalização da solução obtida. Dentro deste espírito, vamos explorar um pouco a solução obtida para o problema descrito a seguir.

Exemplo 5 (Sub-lista de números pares): Dada uma lista xs, desejamos descrever uma sublista de xs formada apenas pelos números pares existentes em xs.

Solução: Devemos considerar a existência de suas situações, como no problema de encontrar a sublista dos menores (Exemplo 4):

1. O primeiro elemento da lista é um número par, neste caso a sublista resultante é dada pela junção do primeiro com a sublista de pares existente no resto da lista.
2. O primeiro não é par. Neste caso a sublista de pares em xs é obtida pela seleção dos elementos pares do resto de xs.

Concluindo, tomemos como base da recursão a lista vazia, que obviamente não contém qualquer número.

Eis a solução em HUGS:

```
--
-- sublista de números pares
--
slpares xs = if null xs
             then xs
             else if even (head xs)
                  then head xs : slpares (tail xs)
                  else slpares (tail xs)
```

E a avaliação de algumas instâncias:

```
Main> slpares [1..10]
[2,4,6,8,10]
(322 reductions, 438 cells)
Main> slpares [1,3..100]
[]
(962 reductions, 1183 cells)
```

Vamos agora, seguindo as orientações do mestre Polya, buscar oportunidades de generalização para esta função. Podemos fazer algumas perguntas do tipo:

1. Como faria uma função para determinar a sublista dos números ímpares a partir de uma dada lista?
2. E se quiséssemos a sublista dos primos?
3. E que tal a sublista dos múltiplos de cinco?

Uma breve inspeção na solução acima nos levaria a entender que a única diferença entre as novas funções e a que já temos é a função que verifica se o primeiro elemento satisfaz uma propriedade, no caso presente a de ser um número par (even), conforme destacamos a seguir:

```
--
-- sublista de números pares
--
slpares xs = if null xs
             then xs
             else if even (head xs)
                  then head xs : slpares (tail xs)
                  else slpares (tail xs)
--
-- sublista de números impares
--
slimpar xs = if null xs
             then xs
             else if odd (head xs)
                  then head xs : slimpar (tail xs)
                  else slimpar (tail xs)
--
-- sublista de números primos
--
slprimo xs = if null xs
             then xs
             else if primo (head xs)
                  then head xs : slprimo (tail xs)
                  else slprimo (tail xs)
```

Isto nos sugere que a função avaliadora pode ser um parâmetro. Pois bem, troquemos então o nome da função por um nome mais geral e adicionemos à sua interface mais uma parâmetro. Este parâmetro, como sabemos, deverá ser do tipo:

```
alfa -> Boolean
```

Vejamos então o resultado da codificação, onde a propriedade a ser avaliada se converte em um parâmetro:

```
--
-- sublista de elementos de xs que satisfazem
-- a propriedade prop
--
sublista prop xs  = if null xs
             then xs
             else if prop (head xs)
                  then head xs : sublista prop (tail xs)
                  else sublista prop (tail xs)
```

Vejamos então algumas aplicações na nossa função genérica para determinar sublistas:

```
Main> sublista even [1..10]
[2,4,6,8,10]

Main> sublista odd [1..10]
[1,3,5,7,9]

Main> sublista (<5) [1..10]
[1,2,3,4]

Main> sublista (>=5) [1..10]
[5,6,7,8,9,10]
```

Observe que a função que havíamos anteriormente definido para determinar os elementos menores que um determinado valor k, da mesma forma que a função para determinar os maiores que k, está contemplada com a nossa generalização. As duas últimas avaliações no quadro acima ilustram a determinação da sublista dos valores menores que 5 e a dos maiores ou iguais a 5.

Exercícios:

Descreva funções em HUGS que utilizem recursão para resolver os problemas abaixo.

1. Obter a interseção de duas listas xs e ys.
2. Dadas duas strings xs e ys, verificar se xs é prefixo de ys.
3. Dadas duas strings xs e ys, verificar se xs é sufixo de ys.
4. Dadas duas strings xs e ys, verificar se xs é sublista de ys.
5. inverter uma lista xs;
6. Definir a função tWhile, que tenha o mesmo comportamento que a função takeWhile.
7. Definir a função dWhile, que tenha o mesmo comportamento que a função dropWhile.
8. Verificar se uma string é um palíndrome (a string é a mesma quando lida da esquerda para a direita ou da direita para a esquerda).
9. Verifique se uma string é uma palavra. Defina uma palavra como formada apenas por letras.
10. Verificar se os elementos de uma lista são distintos.
11. Determinar a posição de um elemento x em uma lista xs, se ele ocorre na lista.
12. Descrever a lista das palavras que existem no texto, dado um texto.
13. Dadas duas listas xs e ys, ordenadas em ordem crescente, obter a lista ordenada resultante da intercalação de xs e ys.
14. Calcular a combinação de uma lista xs, p a p.

17. ORDENAÇÃO RECURSIVA DE DADOS, ALGUNS PROBLEMAS CORRELATOS E ALGUMAS TÉCNICAS DE PROPÓSITO GERAL

17.1 – INTRODUÇÃO: Voltemos então ao problema de ordenar os elementos de uma lista, para o qual já discutimos uma solução no paradigma aplicativo.

Dada uma lista xs desejamos descrever sua ordenação.

Vamos começar propondo e resolvendo um problema mais simples.

Inserção ordenada: Dada uma lista ordenada xs e um elemento k desejamos descrever uma lista a partir de xs, na qual esteja incluído o valor k, com a condição de que a nova lista também esteja ordenada.

Solução : Voltemos a nossa estratégia para obter soluções recursivas. Aqui também temos dois casos:

1. O valor k é menor que o primeiro da lista xs, neste caso a lista resultante é descrita pela junção de k com a lista xs;
2. O valor k é maior ou igual ao primeiro elemento da lista xs, neste caso a lista resultante é descrita pela junção do primeiro da lista xs com a lista obtida pela inserção ordenada de k no resto da lista xs.

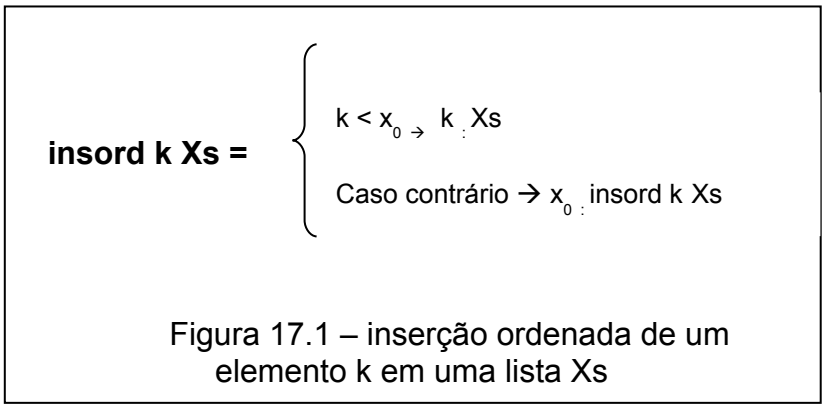


Figura 17.1 – inserção ordenada de um elemento k em uma lista Xs

A codificação pode ser realizada da seguinte forma:

```
--
-- inserção ordenada de um valor k em uma
-- lista ordenada (não decrescente)
--
insord k xs = if null xs
              then [k]
              else if k < head xs
                   then k : xs
                   else head xs : (insord k (tail xs))
```

E a seguir algumas aplicações da solução:

```
Main> insord 5 [0,2..10]
[0,2,4,5,6,8,10]
(230 reductions, 407 cells)
Main> insord 5 [10,15..50]
[5,10,15,20,25,30,35,40,45,50]
(248 reductions, 379 cells)
Main> insord 5 [-10,15..0]
[-10,5]
(92 reductions, 135 cells)
Main> insord 5 [-10,-5..0]
[-10,-5,0,5]
(154 reductions, 220 cells)
Main> insord 5 []
[5]
(23 reductions, 32 cells)
```

Agora já podemos voltar ao nosso problema inicial de ordenação de listas. Vamos em busca de uma primeira solução:

Solução : A ordenação não decrescente de uma lista xs qualquer é igual à inserção ordenada do primeiro da lista na ordenação do resto da lista.

```
--
-- ordenação de uma lista
--
ordena xs = if null xs
            then xs
            else insord (head xs) (ordena (tail xs))
```

Vejamos a aplicação da solução à algumas instâncias:

```
Main> ordena [3, 4, 50,30,20,34,15]
[3,4,15,20,30,34,50]
(241 reductions, 330 cells)
Main> ordena [100,93..50]
[51,58,65,72,79,86,93,100]
(568 reductions, 780 cells)
```

17.2 DIVISÃO E CONQUISTA (UMA TÉCNICA PODEROSA): Alguns problemas possuem soluções mais facilmente descritas, algumas até mais eficientes, quando quebramos o problema em partes menores, descrevemos a solução de cada parte e depois combinamos as soluções parciais para obter a solução completa. Este método é denominado de "divisão e conquista". Basicamente buscamos encontrar instâncias do problema onde a solução seja imediata. . Nesta seção veremos alguns exemplos desta abordagem. O primeiro deles, a pesquisa binária, trata da busca de um elemento em uma lista ordenada. Os outros dois, mergesort e quicksort, apresentam soluções alternativas para a ordenação de uma lista.

17.2.1. PESQUISA BINÁRIA - Voltemos a um problema já apresentado anteriormente, verificação da ocorrência de um elemento a uma lista. Segundo a definição que apresentamos para a função ocorre, apresentada no exemplo 3 do Capítulo 16. Podemos constatar que para avaliar expressões onde o elemento procurado não ocorre na lista, o avaliador de expressões precisará fazer uma quantidade de comparações igual ao comprimento da lista considerada. Na média de um conjunto de avaliações, considerando as avaliações de expressões em que o elemento procurado está na lista, e que a cada vez estaremos procurando por um elemento distinto, teremos um número médio de comparações da ordem de (**n / 2**). Se n for muito grande ficaremos assustados com o número de comparações. Por exemplo, para uma lista de 1000000 (um milhão) de elementos, em média teremos que fazer 500 mil comparações.

Se pudermos garantir que a lista está ordenada, então podemos fazer uso de uma estratégia já discutida anteriormente para reduzir este número. Estamos falando da árvore binária de pesquisa. A estratégia que usaremos consiste em, a cada passo de redução, abandonarmos metade da lista considerada a partir da comparação de k com o elemento que se encontra na metade da lista. Se o elemento buscado (k) for igual ao elemento central, então o processo de avaliação está encerrado. Quando isto não ocorre, devemos então escolher em qual lista devemos procurá-lo. Quando ele é menor que o elemento central devemos buscá-lo na sublista que antecede o central, caso contrário devemos buscá-lo na sublista dos seus sucessores. Novamente a base da recursão é determinada pela lista vazia.

Nesta abordagem, a cada escolha abandonamos metade da lista restante. Desta forma, o número de comparações é dado pelo tamanho da seqüência:

n/1, n/2, n/4, ... , n/n

Para simplificar a análise podemos escolher um n que seja potência de 2. Neste caso podemos assegurar que o comprimento da seqüência é dado por:

Log n na base 2

Voltando então ao número de comparações necessárias para localizar um elemento, podemos constatar que em uma lista com 1 milhão de elementos, ao invés das 500 mil comparações da solução anterior, precisaremos no pior caso, de apenas 20 comparações. Isso mesmo, apenas 20. Vamos então à codificação em HUGS:

```
--
-- pesquisa binária
--
pesqbin k xs = if null xs
                then False
                else if k == pivot
                     then True
                     else if k < pivot
                          then pesqbin k menores
                          else pesqbin k maiores
               where
                 p       = div (length xs) 2
                 menores = take p xs
                 maiores = tail (drop p xs)
```

```
                pivot   = head (drop p xs)
```

E a seguir, a avaliação de algumas instâncias:

```
Main> ocorre 1023 [0..1023]
True
(24592 reductions, 32797 cells)
Main> pesqbin 1023 [0..1023]
True
(71563 reductions, 92060 cells)
```

17.2.2 MERGESORT - Existem outras maneiras de se descrever a ordenação de uma lista. Uma delas, denominada ***mergesort***, se baseia na intercalação de duas listas já ordenadas. Comecemos então por discutir a intercalação que, em si mesmo, já representa uma ferramenta intelectual bastante interessante.

Intercalação: Antes de ver o mergesort podemos apresentar uma versão recursiva para a intercalação de duas listas em ordem não decrescente.

Solução: A intercalação de duas listas ordenadas ***xs*** e ***ys*** pode ser descrita através de dois casos:

1. se o primeiro elemento de ***xs*** é menor que o primeiro elemento de ***ys*** então a intercalação é dada pela junção do primeiro elemento de ***xs*** com a intercalação do resto de ***xs*** com ***ys***;
2. caso contrário, a intercalação é descrita pela junção do primeiro elemento de ***ys*** com a intercalação do resto de ***ys*** com ***xs***;

A codificação resultante pode ser observada a seguir:

```
--
-- Intercala duas listas em ordem não decrescente
--
intercala xs ys = if (null xs) || (null ys)
                  then xs ++ ys
                  else if head xs <= head ys
                       then head xs : intercala (tail xs) ys
                       else head ys : intercala  xs (tail ys)
```

E a seguir, algumas submissões e avaliações do HUGS:

```
Main> intercala [1,3..10] [0,2..10]
[0,1,2,3,4,5,6,7,8,9,10]
Main> intercala [0,2..10] [1,3..10]
[0,1,2,3,4,5,6,7,8,9,10]
Main> intercala [0,2..10] []
[0,2,4,6,8,10]
Main> intercala [] []
ERROR: Unresolved overloading
```

```
*** Type        : Ord a => [a]
*** Expression : intercala [] []

Main> intercala [] [0,2..10]
[0,2,4,6,8,10]
Main> intercala [0,2..10] [0,2..10]
[0,0,2,2,4,4,6,6,8,8,10,10]
Main> intercala [9,7..1] [10,8..1]
[9,7,5,3,1,10,8,6,4,2]

 (o que houve que não ficou ordenada?)
```

Voltemos ao mergesort, ou, em bom português, ordenação por intercalação.

Solução : A ordenação de uma lista por mergesort é igual à intercalação do mergesort da primeira metade da lista com o mergesort da segunda metade. Esta solução explora a noção de árvore binária. Neste caso, a lista original é dividida em 2 partes, cada uma delas em outras duas e assim sucessivamente até que esta quebra não seja mais possível. A figura Fig. 17.1 ilustra o processamento da ordenação de uma lista.

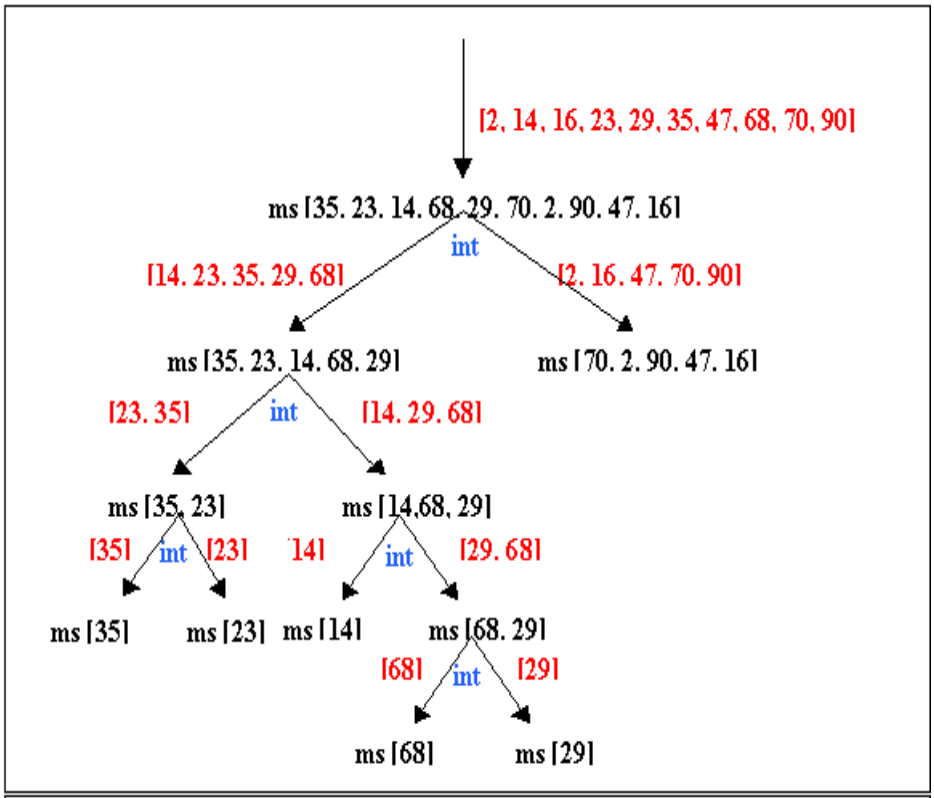


Fig 9.1 : Desenvolvimento da ordenação de uma lista utilizando o mergesort. As lista em preto são as entradas para a função e as listas em vermelho são os valores retornados. Em azul indicamos a função de intercalação (int) aplicada ao resultados de mergesort.

Vejamos então como fica a codificação em HUGS.

```
--
-- ordena uma lista pela intercalação da ordenação de
-- suas duas metades
--
mergesort xs = if null (tail xs)
               then xs
               else intercala (mergesort m) (mergesort n)
               where
                  m = take k xs
                  n = drop k xs
                  k = div (length xs) 2
```

```
Main> mergesort [1..10]
[1,2,3,4,5,6,7,8,9,10]
(1593 reductions, 2185 cells)
Main> mergesort [10,9..1]
[1,2,3,4,5,6,7,8,9,10]
(1641 reductions, 2236 cells)
```

17.2.3. QUICKSORT - Existe uma maneira muito famosa de resolver o mesmo problema de ordenação, usando ainda a noção de divisão e conquista, muito parecida com o mergesort. Implementações desta solução reduzem sensivelmente o número de comparações necessárias e são, portanto muito utilizadas.

Solução : Na versão usando o mergesort dividíamos a instância original exatamente ao meio. Nesta vamos dividir também em duas, mas com seguinte critério: a primeira com os elementos menores que um elemento qualquer da lista e a segunda com os elementos maiores ou iguais a ele. Este elemento é denominado ***pivot*** e existem várias formas de escolhê-lo. A melhor escolha é aquela que produzir as sublistas com comprimentos bem próximos, o que repercutirá no desempenho da avaliação. Aqui nos limitaremos a escolher como pivot o primeiro elemento da lista. Assim sendo, após obter a ordenação das duas listas, basta juntar a ordenação da primeira, com o pivot e finalmente com a ordenação da segunda. A figura Fig. 9.2 ilustra a aplicação do quicksort a uma instância do problema.

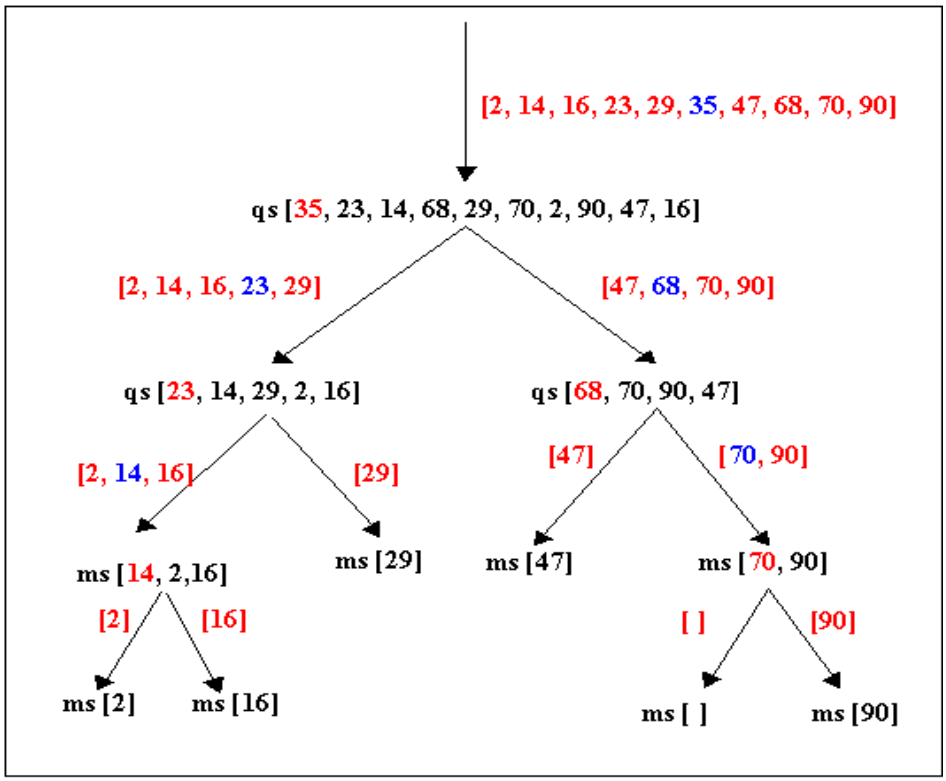


Fig 9.2 : Desenvolvimento da ordenação de uma lista utilizando o quicksort. As listas em preto são as entradas para a função e as listas em vermelho são os valores retornados. Nas listas de entrada indicamos o pivot em vermelho, nas listas de retorno o pivot está indicado em azul.

E a seguir vejamos a codificação.

```
quicksort xs = if  (null xs) || (null (tail xs))
               then xs
               else    quicksort (sublista (<  pivot) (tail xs))
                    ++ [pivot]
                    ++ quicksort (sublista (>= pivot) (tail xs))
               where
                  pivot = head xs
```

Convidamos o leitor a apreciar e discutir a elegância, a compacidade e a clareza da descrição do quicksort.

Vejamos a avaliação de algumas instâncias:

```
Main> quicksort [4,5,6,7,8,3,2,1]
[1,2,3,4,5,6,7,8]
(595 reductions, 755 cells)
Main> quicksort [1..10]
[1,2,3,4,5,6,7,8,9,10]
(1536 reductions, 1881 cells)
Main> quicksort [10,9..1]
[1,2,3,4,5,6,7,8,9,10]
(1541 reductions, 1952 cells)
```

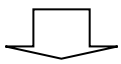
```
Main> mergesort [10,9..1]
[1,2,3,4,5,6,7,8,9,10]
(1647 reductions, 2283 cells)
Main> mergesort xs == quicksort xs where xs = [1..10]
True
(2805 reductions, 3563 cells)
Main> mergesort [2,14,16,23,29,35,47,68,70,90]
[2,14,16,23,29,35,47,68,70,90]
(1414 reductions, 1922 cells)
Main> quicksort [2,14,16,23,29,35,47,68,70,90]
[2,14,16,23,29,35,47,68,70,90]
(1357 reductions, 1618 cells)
Main> ordena [2,14,16,23,29,35,47,68,70,90]
[2,14,16,23,29,35,47,68,70,90]
(236 reductions, 336 cells)
```

17.3. PROCESSAMENTO DE CADEIAS DE CARACTERES: As cadeias de caracteres, como já vimos, também são listas, portanto o uso de recursão com cadeias segue as mesma recomendações. Para ilustrar vamos apresentar alguns exemplos.

Exemplo 01 - [Palíndromo] Dada uma cadeia de caracteres verifique se é um palíndromo. Segundo o dicionário, um palíndromo é uma frase ou palavra, que não importando o sentido que se lê, significa a mesma coisa. Por exemplo, "Socorram-me subi no Ônibus em Marrocos". Vejam que a quantidade de espaços, os separadores e os términos de palavra não são considerados. Aqui vamos tratar a questão de forma simplificada, os separadores serão tratados como caracteres comuns.

Solução : Neste caso, é importante observar que podemos olhar a cadeia como sendo formada por pares de valores eqüidistantes dos extremos. Um cadeia é palíndromo se os seus extremos são iguais e o meio da lista é um palíndromo. A base da recursão são as cadeias vazias ou aquelas com apenas um elemento.

palíndromo $[x_0, x_1, x_2, x_3, x_4, x_5, ..., x_{n-1}]$



$x_0 = x_{n-1}$ & palíndromo $[x_1, x_2, x_3, x_4, x_5, ...,]$

$x_0 =$ & $(x_1 = x_{n-1}$ & palíndromo $[x_2, x_3, x_4, x_5, ...,])$

$x_0 =$ & $(x_1 = x_{n-1}$ & $(X_2 = X_{n-2}$ & palíndromo $[x_2, x_4, x_5, ...,]))$

Figura 17.1 – desenvolvimento da computação da função palíndromo

Vejamos então a codificação em HUGS e a avaliação para algumas instâncias.

E agora uma avaliação de algumas listas candidatas a palíndromo:

```
--
-- Verifica se uma sentença é Palíndromo
--
--
palindromo xs = if null xs || null (tail xs)
                then True
                else (head xs == last xs) &&
                     palindromo (meio xs)
                where
                   meio xs = init (tail xs)
```

Seguindo nosso padrão de apreentação,vejamos como ficam algumas avaliações:

```
Main> palindromo "socorrammesubinoonibusemmarrocos"
True
(687 reductions, 698 cells)
Main> palindromo "socorram-me subi no onibus em marrocos"
False
(891 reductions, 907 cells)
Main> palindromo "ama"
True
(31 reductions, 43 cells)
Main> palindromo "papagaio"
False
(29 reductions, 45 cells)
Main> palindromo "arara"
True
(49 reductions, 61 cells)
```

Exemplo 02 - [Prefixo] Dadas duas cadeias de caracteres verifique se a primeira é idêntica à subcadeia formada pelos primeiros caracteres da segunda. Por exemplo, "aba" é prefixo da cadeia "abacaxi" e "pre" é prefixo de "prefixo".

Solução : De imediato podemos dizer que uma cadeia xs é prefixo de uma cadeia ys quando:

iii) o primeiro elemento de xs é igual ao primeiro elemento de ys e;
iv) o restante de xs é prefixo do restante de ys.
v)

Quanto à base da recursão, temos que considerar duas situações:

i) A primeira tem como base que a cadeia vazia é prefixo de qualquer outra cadeia;
ii) A segunda leva em conta que nenhuma cadeia pode ser prefixo de uma cadeia vazia (exceto a cadeia vazia).

Vejamos como fica em Haskell:

```
--
-- Verifica se uma cadeia xs é prefixo
-- de uma segunda (ys)
--
prefixo xs ys = if null xs
                then True
                else if null ys
                     then False
                     else (head xs == head ys) &&
                          prefixo (tail xs) (tail ys)
```

As avaliações de expressões a seguir nos permitem observar o funcionamento de nossa descrição:

```
Main> prefixo "aba" "abacadraba"
True

Main> prefixo "" "abacadraba"
True
Main> prefixo "pre" "prefixo"
True

Main> prefixo "prefixo" "pre"
False

Main> prefixo "prefixo" ""
False
```

Exemplo 03 - [Casamento de Padrão] Verificar se uma cadeia satisfaz um determinado padrão é um processamento muito útil e constantemente realizado na prática da computação. Aqui nos ateremos a uma forma simplificada deste problema que consiste em verificar se uma cadeia é ***subcadeia*** de outra.

Solução : Uma rápida inspeção nos leva à constatação de que o problema anterior é parecido com este, exceto pelo fato de que a primeira cadeia pode ocorrer em qualquer lugar da segunda. Podemos dizer então que a primeira cadeia ocorre na segunda se ela é um prefixo da primeira ou se ela ocorre no resto da segunda.

Vejamos como fica em HUGS:

```
--
-- Verifica se uma cadeia xs é subcadeia
-- de uma outra (ys)
--
subcadeia xs ys = if null ys || null (tail ys)
                  then False
                  else prefixo xs ys || subcadeia xs (tail ys)
```

A avaliação das expressões a seguir ajuda no entendimento:

```
Main> subcadeia "" "prefacio"
True
Main> subcadeia "pre" "prefacio"
True
Main> subcadeia "cio" "prefacio"
True
Main> subcadeia "efa" "prefacio"
True
Main> subcadeia "acido" "prefacio"
False
Main> subcadeia "efa" ""
False
```

Exercícios:

1. Pesquise como é descrito o método da “bolha” para ordenação. Faça uma função em Haskell para ordenar uma lista pelo método da bolha.

18. APLICAÇÕES

Neste capítulo apresentamos uma série temática de exercícios, buscando dar ao leitor uma visão mais ampla das possibilidades de uso da programação funcional. A intenção é apresentar vários contextos onde precisamos programar computadores. Em todos, o contexto é descrito e vários exercícios são apresentados. Aos professores e estudantes, sugerimos que o contexto sirva de pretexto para a formulação e resolução de novos exercícios. Começamos pelo **Dominó Baré**, onde, buscamos na ludicidade do jogo, apresentar as necessidades de manipular informação. Na seqüência apresentamos uma série de problemas considerando as necessidades de informação de um **Banco de Sangue**, o manuseio de mensagens de um **Correio Eletrônico**, a organização de uma **Lista de Compras**, um sistema de **Passagens Aéreas, Gerência Acadêmica, Agência de Turismo** e exercícios sobre **Espetáculos Teatrais**.

18.1 O DOMINÓ BARÉ: O dominó de números é uma coleção de pedras, utilizado na maioria das vezes como um excelente passatempo. Das tantas formas de utilizar o dominó, destacamos uma, utilizada no Amazonas, principalmente em Manaus, mas também em muitas praias pelo mundo afora onde existirem amazonenses, em particular nas praias de Fortaleza. Os Amazonenses costumam chamá-la de “dominó baré", em homenagem a uma tribo que habitava a região onde foi fundada a cidade de Manaus. A maioria dos exercícios deste capítulo foram desenvolvidos em Manaus, no final da década de 80. De lá pra cá, vários outros foram acrescentados, mas a lista como um todo permanece inédita.

A intenção desta seção é apresentar alguns poucos exercícios resolvidos e propor outros. A idéia não é desenvolver o jogo e sim, inspirar-se em situações do jogo para propor exercícios interessantes e desafiadores. Deixamos o desenvolvimento do jogo completo como sugestão para o trabalho em grupos. Ao final do capítulo discutiremos um pouco sobre a programação do jogo.

Preliminares: O material do jogo é um conjunto formado por 28 "peças", cada uma delas com duas "pontas". Cada "ponta" representa um valor de 0 (zero) a seis (6), perfazendo portanto 7 valores diferentes. Cada valor possui um nome próprio: o Zero chama-se "branco", o um chama-se "ás", o dois é o "duque", o três chama-se "terno", o quatro é a "quadra", o cinco é a "quina" e o seis denomina-se "sena". O nome de uma "pedra" é dado pelo nome de suas "pontas", por exemplo, "quina e terno", é o nome da "pedra" que possui em uma ponta o valor 5 e na outra o valor 3. As pedras que possuem o mesmo valor nas duas pontas são denominadas de "carroça". Para cada tipo de valor existem 7 pedras, por exemplo, para o "terno" teremos: terno e branco, terno e ás, terno e duque, carroça de terno, quadra e terno, quina e terno, sena e terno. O jogo é, em geral, disputado por duplas, ganhando a que fizer o maior numero de pontos, a partir de um mínimo pré-estabelecido. A seguir apresentamos em detalhes os vários elementos do jogo.

Peças do Jogo: Os elementos do jogo são (28) vinte e oito peças, cada uma com duas pontas, na qual é marcado um valor que varia de 0 a 6. Para jogar, as "pedras" são embaralhadas e escolhidas pelos jogadores. A cada jogador cabem 7 pedras. Com o desenrolar do jogo a quantidade de pedras vai sendo decrescida, até que, eventualmente chegue em zero.

Participantes: duas duplas (eventualmente pode ser jogado individualmente, com 2, 3 ou 4 jogadores).

Objetivo: atingir um total mínimo de 200 pontos. Vence o jogo a dupla que ao final de uma rodada tiver o maior número de pontos.

Dinâmica: o jogo se desenvolve em uma quantidade qualquer de eventos menores denominados de rodada. A figura 18.1 ilustra um instante de jogo.

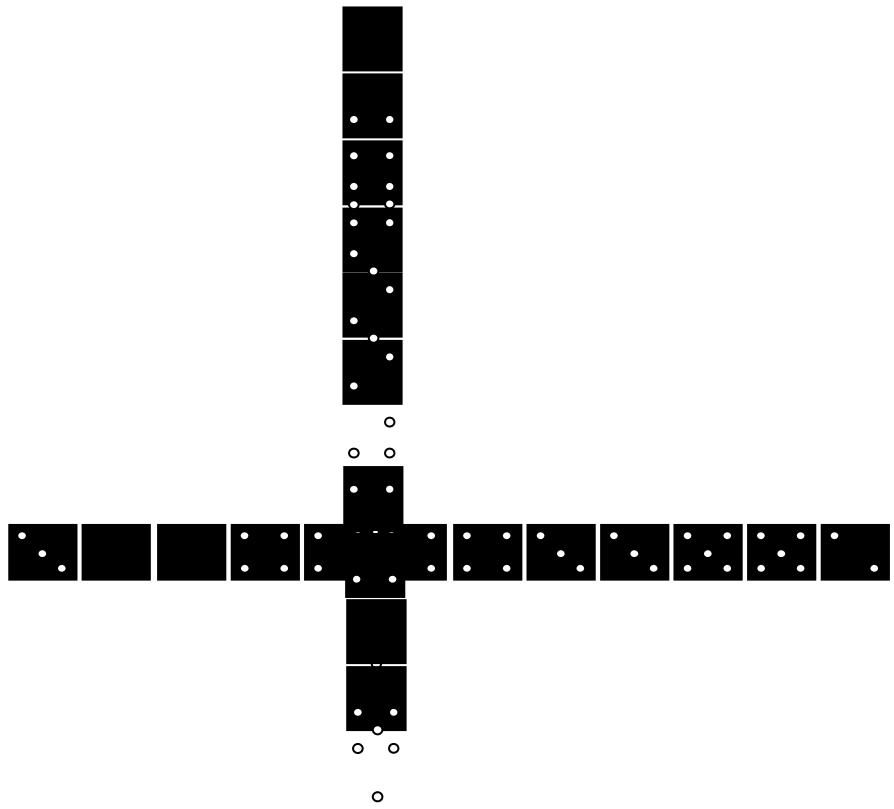


Figura 18.1 – um instante do jogo de dominó, com quatro pontas abertas: terno, às, duque e quadra.

Rodada: em uma rodada, um após o outro, no sentido horário, os jogadores vão fazendo suas jogadas, combinando suas pedras de dominó com a “figura” que já está formada na mesa de jogo.

Pontuação: existem 4 formas para obter pontos:

1. Durante o jogo, a figura formada na mesa possui 1 (quando existe apenas uma peça assentada), 2, 3 ou 4 pontas. Á soma dos valores dessas pontas denomina-se de: “os pontos da mesa”. Quando essa soma produz um múltiplo de 5, o jogador que sentou a última pedra pode requerer que eles sejam anotados em favor de sua

dupla. Veja que só o jogador que sentou a pedra pode reivindicar os pontos e isto tem que ocorrer antes que o próximo jogador sente a sua pedra;

2. Quando um jogador não possui pedra para colocar na mesa (ou seja, uma que combine com uma das pontas), ele passa a vez, e a dupla adversária ganha 10 pontos. Se um jogador percebe que com a colocação de sua peça ele conseguirá fazer com que todos os demais passem, inclusive o seu parceiro, ele pode anunciar que deu um passe geral e com isso ganhar de bônus 50 pontos.
3. Quando um jogador descarta sua última peça em uma rodada diz-se que ele "bateu", e, portanto ganhou a rodada. Com isso ele ganha de bônus 10 pontos e mais o múltiplo de 5 ligeiramente inferior à soma dos valores constantes nas peças que sobraram nas mãos dos adversários (garagem). Se a batida for feita com uma carroça, o bônus é de 20 pontos.
4. Quando ocorre uma situação onde nenhum dos jogadores consegue jogar, embora estejam com peças na mão, diz-se que o jogo está fechado. Neste caso ganha a rodada a dupla cuja soma dos valores das peças for o menor. A soma das peças da dupla adversária é computada em seu favor, como no caso 3.

Posição dos Jogadores: Os membros de cada dupla são colocados em posições alternadas, de forma que as jogadas (colocação de peças) seja feita de forma alternada entre as duplas adversárias. Por exemplo, em um jogo presencial, podemos usar, como suporte para colocação das peças, uma mesa de quatro lugares, ficando os parceiros sentados frente-a-frente.

Distribuição das Peças: a distribuição das 28 peças entre os 4 jogadores deve ser feita de forma aleatória. Na prática, em um jogo com peças físicas, viram-se as peças de cara para baixo e mistura as peças com as mãos. Cada jogador vai retirando as suas próprias peças.

Quem começa uma rodada: Uma rodada é sempre iniciada com a colocação de uma "carroça". Na primeira rodada do jogo, a carroça a ser utilizada é de sena (6), cabendo pois ao jogador que a tirou começar o jogo. As rodadas seguintes são iniciadas pelo jogador que bateu a rodada anterior, com a carroça que ele preferir. Se ele não possui carroça, ele passa e o jogador seguinte (da dupla adversária) inicia o jogo, se este também não possuir, passa a frente.

Uma Jogada: Estando na sua vez de jogar, o jogador deve escolher uma das pedras de sua mão, e coloca-la na mesa de jogo, combinando com alguma das pontas abertas. A escolha da peça a ser jogada deve contribuir para o objetivo da dupla que é ganhar o jogo, isso significa, em linhas gerais, escolher uma pedra que me permita fazer o maior número de pontos e, quando isso não for possível, escolher uma pedra que reduza o número de pontos que os adversários possam fazer com base em minha jogada. Há, entretanto algumas nuances a serem consideradas:

- Quando eu jogo fazendo pontos devo buscar maximizar meus pontos e minimizar os que o jogador adversário possa fazer a partir da minha jogada;
- Se o jogo estiver próximo do término, e a dupla adversária ameaça completar os 200 pontos, pode ser desejável adiar o término, não fazendo os pontos. Por exemplo, suponha que a dupla adversária tem 185 pontos e a minha 130. Se eu tiver uma peça na mão que faz 25 pontos, mas se é possível ao adversário possuir

uma peça que lhe permita fazer 15 pontos, eu posso escolher outra peça, deixando assim de obter os 25 pontos;

- Quem abre uma rodada, baterá a rodada a menos que passe. Tendo em vista que ao ganhar uma rodada, há bônus para a dupla, posso deixar de marcar ponto visando vencer a rodada;
- Idem para tentar evitar que a dupla adversária ganhe a rodada (quando forço a passada de um adversário que começou a rodada, a batida passa para o meu parceiro).
- Um passe geral dá um bônus de 50 pontos, isso pode me levar a buscá-los, desde que as condições do jogo, definidas pelas peças de minha mão, combinadas com o que já foi jogado, se mostrem propícias.

Exercícios: A seguir apresentamos alguns exercícios, baseados no dominó. Para fins didáticos separamos em grupos.

Grupo I

1. Escreva a função ***pedrap*** que associe um par a **True** se e somente se (sss) o par é uma representação válida para uma "pedra" e **False** caso contrário.

   Exemplos de uso:

   1. pedrap (2, 7) ==> **False**
   2. pedrap ((-3), 4) ==> **False**
   3. pedrap (3,4) ==> **True**

   Solução:

```
pedrap (x,y) = validap x && validap y
validap x    = elem x [0..6]
```

2. Escreva a função **maop** que associe uma lista de pares de inteiros a **True** sss a lista é uma representação válida para a "mão" de um jogador e **False** caso contrário.

   Exemplos de uso:

   1. maop [ ] ➔ **True**
   2. maop [((-3), 4)] ➔ **False**
   3. maop [(3,4)] → **True**
   4. maop [(1,2), (1,5), (2,0), (2,4), (3,3), (1,1), (0,0), (4,0)] ➔ **False**

   Solução:

```
maop [ ]      = True
maop (x:xs) = (length xs <= 6) && pedrap x && maop xs
```

3. Escreva a função **carrocap** que associe um par a **True** sss o par é uma "carroça" e **False** caso contrário.
4. Escreva a função **tem_carroca_p** que associe uma "mão" a **True** sss a mão possuir pelo menos uma carroça e **False** caso contrário.
5. Escreva a função **tem_carrocas** que associe a uma "mão" a lista das "carroças" nela contida.

Grupo II

Em vários momentos do jogo faz-se necessário saber a quantidade de pontos associado à uma coleção de pedras. Em particular, no final do jogo, quem "sentou" a sua última pedra faz jus à "garagem" que é determinada a partir dos pontos que restaram na(s) mão(s) dos adversários.

6. Escreva a função ***pontos*** que associe uma lista de "pedras" a soma dos pontos das pedras nela contidos. Onde os pontos de uma pedra é a soma de suas pontas.

   Solução:

```
pontos [ ]      = 0
pontos (x:xs) = ponto x + pontos xs
                  where
                     ponto (x,y) = x + y
```

7. Escreva a função ***garagem*** que associe uma lista de "pedras" ao maior múltiplo de 5 (cinco), menor ou igual à soma dos pontos nela contidos.
8. Escreva a função ***pedra_igual_p*** que associe dois pares de inteiros a **True** sss representam a mesma pedra e **False** caso contrário. É bom lembrar que a ordem das pontas é irrelevante, assim (2,4) e (4,2) representam a mesma pedra.
9. Escreva a função ***ocorre_pedra_p*** que associe uma "pedra" e uma "mão" a **True** sss a "pedra" ocorre na "mão" e **False** caso contrário.
10. Escreva a função ***ocorre_valor_p*** que associe um valor válido para "ponta" e uma "mão" e produza **True** sss o valor ocorre em alguma pedra da mão e **False** caso contrário.
11. Escreva a função ***ocorre_pedra*** que associe a um valor e uma "mão", uma lista contendo as pedras da "mão" que possuem o valor dado.
12. Escreva a função ***pedra_maior*** que associe uma "mão" a pedra de maior valor na "mão" dada. Uma pedra p1 é maior que uma outra p2 sss a soma das pontas de p1 for maior que a soma das pontas de p2.
13. Escreva a função ***ocorre_valor_q*** que associe um valor e uma "mão" e produza o número de pedras na mão que possuem o valor dado.
14. Escreva a função ***ocorre_carroca_q*** queassocie uma mão à quantidade de carroças nela existentes.
15. Escreva a função ***tira_maior*** que associe uma mão a uma lista similar à "mão" de onde foi extraída a pedra de maior ponto.

16. Escreva a função ***tira_maior_v*** que associe um valor e uma "mão" à lista similar à "mão" de onde se extraiu a pedra de maior pontos de um determinado valor para ponta.

Grupo III

O jogo se desenvolve pela colocação, pelo Jogador da vez, de uma pedra que combine com alguma das "pontas" da "mesa". Num momento genérico do jogo temos quatro pontas disponíveis para execução de uma jogada. Uma ponta pode ser simples ou uma carroça. As carroças são dispostas de tal forma que todos os seus pontos estejam para "fora".

Chamaremos "mesa" à lista de pontas disponíveis para jogada. Pontas simples serão representadas por listas de um elemento e carroças por uma lista com dois elementos idênticos. Por exemplo, a "mesa" ilustrada na Figura 18.2 é representada pela quádrupla ([5,5], [5], [0],[4]).

Uma ponta ainda não aberta é representada por lista vazia. Dizemos que há marcação de pontos em uma mesa quando a soma das pontas é um múltiplo de 5. Os pontos a serem marcados é a soma das pontas, com as carroças contando em dobro.

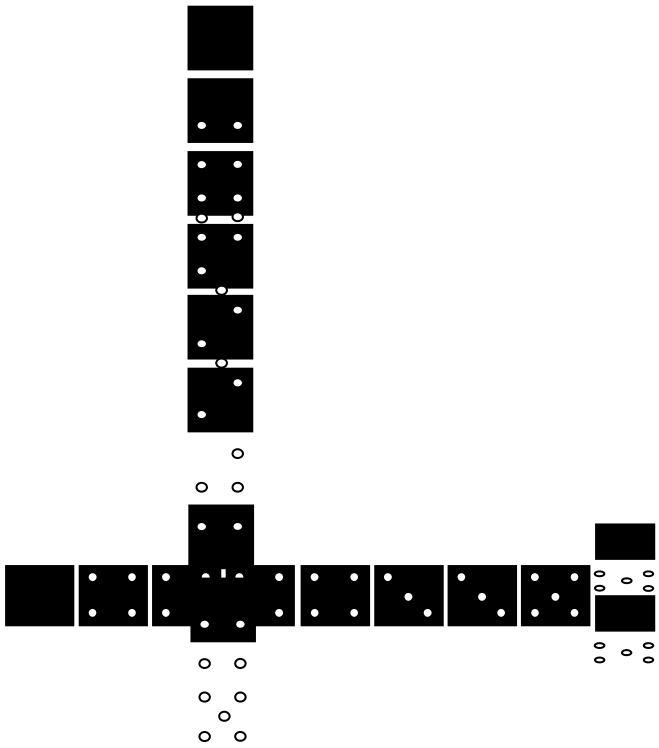


Figura 18.2 – desenvolvimento do jogo, no instante em que temos nas pontas externas, uma carroça de quina, uma quina, branco e quadra.

17. Escreva a função ***mesap*** que associe uma quádrupla de listas a **True** sss a quádrupla for uma descrição válida de "mesa".

    Solução:

```
mesap (p1,p2,p3,p4) =  vponta p1 && vponta p2 &&
                       vponta p3 && vponta p4
                                                          where
                            vponta (x:y:xs) = if not (null xs)
                                              then False
                                              else validap x && vponta (y:xs)
                            vponta (x :[ ] )  = validap x
                            validap x         = elem x [0..6]
```

18. Escreva a função ***carroca_m_p*** que associe uma mesa a **True** sss pelo menos uma das pontas for carroça.
19. Escreva a função ***pontos_marcados*** que associe uma mesa ao o número de pontos a serem marcados se a soma das pontas for múltiplo de cinco e zero em caso contrário.
20. Escreva a função ***pode_jogas_p*** que associe uma "pedra" e uma "mesa" a **True** sss a pedra possui uma ponta que combina com pelo menos uma das pontas da mesa.
21. Escreva a função ***marca_ponto_p*** que tenha como entrada uma "pedra" e uma "mesa" e produza **True** sss a pedra pode ser jogada fazendo pontos em uma das pontas da mesa. Lembre-se que as carroças devem ser contadas pelas duas pontas da pedra.
22. Escreva a funão ***maior_ponto*** que tenha associa uma pedra e uma mesa ao número da "ponta" da mesa onde pode ser marcado o maior valor de ponto que será marcado pela pedra. Considere que a em uma "mesa" as pontas são numeradas a partir de zero, da esquerda para a direita.
23. Escreva a função ***joga_pedra*** que associe uma "pedra", uma "mesa" e um número de "ponta" da mesa a uma nova mesa obtida ao se jogar a "pedra" na "ponta" indicada.
24. Escreva a função ***jogap*** que associe uma "mão" e uma "mesa" e produza **True** sss existe pelo menos uma pedra na mão que possa ser jogada em pelo menos uma ponta da mesa. Caso contrário produza **False**.
25. Escreva a função jogada que associe uma "mão" e uma mesa ao número da pedra na mão e número da ponta na mesa onde pode ser feita a jogada que marque mais ponto. Considere inclusive jogada onde não há marcação de ponto.
26. Escreva a função faz_jogada que associe uma "mão" e uma "mesa" e produza uma nova "mesa" obtida por se jogar marcando o maior número de pontos possível

18.2 Banco de Sangue: para facilitar o atendimento da demanda por transfusões de sangue o sistema de saúde criou os chamados Bancos de Sangue. Como sabemos, cada transfusão só pode ser realizada usando tipos de sangue apropriados. A adequação de um determinado tipo de sangue é baseada em estudos científicos que identificou quatro tipos

sangüíneos, denominados de ***A, B, AB e O***. Outros estudos identificaram ainda a existência do chamado fator RH que pode ser positivo (+) ou negativo (-), assim o sangue de qualquer indivíduo é classificado de acordo com esses dois atributos. Por exemplo, dizemos que fulano possui sangue tipo O e fator RH positivo, e abreviamos para **O+**. Em um dado Banco de Sangue, diariamente, são feitas doações por pessoas de diferentes tipos sangüíneos, para as quais é feito um registro contendo o número da carteira de identidade do doador (RG), o sexo (S), a data da doação (DD), a data de nascimento (DN), o tipo sangüíneo (TS), o fator RH (RH) e a quantidade doada (QD) (250 ou 500 ml). O sangue doado é guardado em recipientes com uma capacidade fixa (250 ml). Também, diariamente são feitas requisições pelos hospitais (H), cada requisição indica as características do sangue (tipo e fator RH) e a quantidade solicitada (QS). Sabemos que homens e mulheres possuem intervalos de tempo diferentes para fazer doações. Para homens o intervalo mínimo é de 2 (dois) meses e para mulheres é de 3 (três). A idade máxima para doadores é 60 anos.

Sejam as seguintes estruturas

Doação	(RG, S, DD, DN, TS, RH, QD)
Requisição	(H, TS, RH, QS)

Exercícios: Escreva programas em HUGS para resolver os seguintes problemas:

1. Dada uma lista de doações, obtenha a quantidade total de sangue doado por tipo sangüíneo e fator RH. O resultado será uma tupla (um item para cada combinação de tipo sangüíneo com fator RH) com triplas explicitando o tipo sangüíneo, o fator RH e a quantidade total. Quando não houver doação de uma dado par tipo-fator deve ser indicado o valor zero. Por exemplo:

   ((‘A’, ‘+’, 0), ... (‘O’,’+’, 5500) ...)

2. Para uma dada lista de doações, determine a lista dos dias de um dado mês onde as doações foram menores que a média mensal.
3. Dada uma lista de doações e a data atual, determine a lista de doadores que já estão aptos a fazerem novas doações.
4. Dada a lista de doadores e o mês, determinar o número de doadores que estão aptos a doar sangue naquele mês. (Essa e a questão 3 parecem análogas, não?)
5. Determine a relação de doadores que fazem doações com o maior índice de regularidade. O índice de regularidade é dado pela número de vezes que o intervalo entre as doações coincidem, dividido pelo número de doações menos um.
6. Dada a lista de doadores, verificar o tipo sangüíneo que é mais comumente doado.
7. Dada a lista de doadores e o ano, determine o mês em que houve mais doações naquele ano.

8. Dada a lista de requisições de um determinado hospital, determinar a lista de tipos sangüíneos com os respectivos fatores RH, que possuem seus pedidos atendidos pelo banco de sangue.
9. Determinar, para um dado hospital em um determinado ano, a demanda mensal de sangue, por tipo sangüíneo e fator RH.
10. Determinar a lista de doadores que não estão mais aptos a fazer doações, considerando a data atual.
11. Considere o estoque atual do banco de sangue, determinado pela função estoque (prob 1), e uma lista com várias requisições. Leve em conta que o estoque pode ser insuficiente para atender completamente todos os pedidos. Determine o estoque atualizado após o atendimento dos pedidos e produza uma lista das requisições atendidas, constando a quantidade que foi de fato fornecida.
12. No problema 11, considere que você deseja atender cada hospital solicitante, de forma proporcional ao seu pedido, considerando os pedidos de cada tipo sangüíneo separadamente. Por exemplo, suponha que: o total de pedidos de sangue O+ é de 12.000 ml, que o hospital "h1" solicitou 3.000 ml de sangue O+ e que no estoque 8.000 ml. Podemos observar que o pedido para o sangue O+ do hospital "h1" representa 25 % do total. Neste caso o hospital "h1" será atendido com 25 % de 8.000 ml que representa 2.000 ml. Produza uma lista como os pedidos atendidos e outra com os pedidos pendentes.
13. Considere a política de atendimento do problema 12 mas leve em conta que um dado pedido deve ser atendido completamente. Considere o exemplo do problema anterior, e suponha que os pedidos do hospital 'h1' para o sangue O+ são 4, ("h1", O, +, 1.500), ("h1", O, +, 1.000) e ("h1", O, +, 250) e ("h1", O, +, 500). Neste caso, considerando que os pedidos estão em ordem de prioridade, seriam atendidos os pedidos ("h1", O, +, 1.500) e ("h1", O, +, 250).
14. Modifique a política de atendimento do problema 14 para que o atendimento seja tal que o hospital "h1" use da melhor forma possível a proporcionalidade que lhe cabe. No caso do exemplo apresentado no problema 14, o pedido de 250 ml seria descartado visto que atendendo o pedido de 500 ml o hospital 'h1" estará usando melhor a parte que lhe cabe. (escolher a combinação mais apropriada)

18.3 Correio Eletrônico: Considere um sistema de mensagens eletrônicas. Uma mensagem pode ser descrita por uma tupla contendo o remetente, o destinatário, a data de envio, o assunto e o corpo da mensagem. Mensagens podem ser acumuladas em listas para posterior acesso. Para facilitar o acesso podemos construir índices baseados nos dados contidos na mensagem. Por exemplo, podemos ter um índice baseado no remetente para facilitar o acesso a todas as mensagens de um dado remetente. Considere as seguintes estruturas:

mensagem	(remetente, destinatário, data, assunto, corpo)
índice	[(argumento1,[ordem na lista de mensagens]), (argumento2, []), ...]

Exercícios: Elabore programas em HUGS, **usando recursão**, para atender aos seguintes problemas. A interface (nome e parâmetros) das funções é dado em cada uma das questões.

1. (indexa msgs) Dada uma lista de mensagens, produza o índice das mensagens por remetente. Um índice terá a seguinte estrutura: [(remetente1, lista de ocorrências), (remetente2, lista de ocorrências), ...] onde lista de ocorrências é formada pela posição na lista de mensagens onde o remetente ocorre.
   Por exemplo:
   [(“jose@inf.ufes.br”, [1, 23]), (“maria@inf.ufes.br”, [10, 20, 50]), ...]

2. (consulta r p) Considere definidos um índice por remetente, um índice por palavras constantes no assunto das mensagens e uma lista de mensagens. Dados um remetente (r) e uma palavra(p), obtenha a lista de mensagens enviadas por *r* onde a palavra *p* ocorre no assunto.

   remetentes = [(“remetente1”, [...]),(“remetente2”, [...]), ...]
   palav_assunto = [(“palavra1”. [...]),(“palavra2”. [...]), ...]
   mensagens = [*mensagem1, mensagem2, mensagem3, ...*]

3. (msgPmes a r msgs) Dado um ano (*a*), um remetente (*r*) e uma lista de mensagens (*msgs*), verificar a quantidade mensagens enviadas por *r* em cada mês.

4. (busca p ind msgs) Considere um índice construído na forma de lista (*indb*). O primeiro elemento é um par com uma palavra (*p*) e a lista de mensagens (*msgs*) onde *p* ocorre, o segundo elemento é uma lista no mesmo formato de *indb*, para as palavras menores que *p* e o terceiro para as palavras maiores que *p.*

   índice = [(palavra1, [...]), índice para palavras menores que p, índice para palavras maiores que p]

   Quando não houver palavras menores ou maiores que uma dada palavra, o índice é igual a uma lista vazia.
   Dada uma palavra p, o índice (indb) e a lista de mensagens (msgs), descreva a lista mensagens onde p ocorre, usando o índice dado.

5. (palavPassunto msgs) Considere definida uma lista de palavras irrelevantes (lis). Dada uma lista de mensagens (msgs) produza um índice com as palavras distintas que ocorrem nos assuntos das mensagens e não ocorrem em *lis*. Para cada palavra deve ser produzido também a lista das posições na lista de mensagens (msgs) onde ela ocorre.

   lis = [“palavra1”, “palavra2”. ...]

6. (releva msgs li lf) Dada uma lista de mensagens podemos obter uma lista de palavras relevantes. Define-se como palavra relevante em uma lista mensagens (msgs) aquelas

cuja freqüência satisfazem um intervalo para o qual são dados um limite superior (ls) e um limite inferior (li).

7. (constante msgs a) Dada uma lista de mensagens (*msgs*), determinar a lista de remetentes que enviaram pelo menos uma mensagem para cada mês de um dado ano *a*.

8. (freqData msgs m) Para uma dada lista de mensagens desejamos obter a quantidade de mensagens para cada dia de um dado mês *m*.

9. (identico indb1 indb2) Dados dois índices no formato de árvore binária de pesquisa desejamos verificar se são idênticos. Dizemos que dois índices são idênticos quando a palavras e as listas de ocorrência coincidem e os subíndices das palavras menores e o das palavras maiores respectivamente são idênticos.

   índice = ((palavra1, [...]),
   índice para palavras menores que p, índice para palavras maiores que p)

10. (palavOrd indb) Dado um índice no formato de árvore binária de pesquisa produza uma lista das palavras nele contidas de tal forma que as palavras se apresentem em ordem alfabética crescente.

11. (resgate indb) Dado um índice no formato de árvore binária de pesquisa produza uma lista das palavras que ocorrem em cada mensagem. A lista resultante terá o seguinte formato:
    [[palavras da mensagem de ordem 0], [palavras da mensagem de ordem 1], ...]

12. (balance arbinpq) Uma árvore binária de pesquisa está balanceada se e somente se a quantidade (q1) de elementos no subindice das palavras menores difere da quantidade (q2) de elementos no subindice das palavras maiores de no máximo um (1) e q1 é maior ou igual a q2.

$$q2 + 1 >= q1 >= q2$$

13. (insOrd indb msg) Dados um índice (*indb*) no formato de árvore binária de pesquisa e uma mensagem (*msg*), descreva a nova árvore obtida pela inserção das palavras das mensagem (*msg*), exceto as irrelevantes.

14. (perfil msg diret fga) Um diretório é uma lista de assuntos, cada um dos quais associado a uma coleção de palavras. Dada uma mensagem e um diretório podemos atribuir à mensagem um perfil que é uma lista de valores indicando o grau de aproximação (*ga*) dela com cada assunto. O *ga* de uma mensagem com respeito a um assunto pode ser obtido com base na freqüência com que as palavras a ele associadas ocorrem na mensagem. Considere que a função que calcula o *ga* é fornecida e opera sobre uma lista de inteiros.
    Considere os seguintes formatos:

Diretório	*[(“assunto1”, [palavra1, palavra2, ...]), (“assunto2”, [palavra1, palavra2, ...]), ...]*
Perfil	*[(“assunto1”, ga1), (“assunto2”,ga2), ...]*

18.4 Lista de Compras: Para realizar um determinado projeto necessitamos adquirir certos componentes eletrônicos. No mercado de componentes existem vários fornecedores que vendem seus produtos com preços diferenciados. A escolha da melhor alternativa para satisfação de nossas compras depende de vários fatores, dos quais o melhor preço é um dos mais importantes.

Considere as seguintes definições:

Tabela de Preço	Uma lista contendo todos os preços dos componentes comercializados por um determinado revendedor. Cada elemento da lista é um par no formato (*material*, *preço*). Exemplo: [("potenciometro", 2.50), ("resistor-100k", 0.30), ("capacitor-10mF",0.50), ("indutor-10R",3.00)]
Pedido de Compra	Uma lista contendo todos os materiais necessários para um determinado projeto, com suas respectivas quantidades. Cada elemento da lista é um par no formato (*material*, *quantidade*). Exemplo: [("transformador", 50), ("fonte DC", 10), ("resistor-200k",100)]
Lista de Revendedores	Uma lista contendo a tabela de preços de todos os revendedores, no formato: [(revendedor1, [tabela de preço]), ...]

Exercícios: Elabore programas em HUGS, **usando recursão**, para atender aos seguintes problemas. O nome das funções é dado em cada uma das questões.

1. (custo) Dado o resultado da pesquisa de preços de um pedido de compra, para um certo fornecedor, no formato [(material1, qtde, preço), (material2, qtde, preço), ...], queremos obter o custo total da compra se optarmos por esse fornecedor.
2. (fornece) Dado um pedido de compra e a tabela de preços de um determinado revendedor, obtenha a lista dos materiais que ele pode fornecer.
3. (subprojeto) Considere os pedidos de compra para dois projetos (p1 e p2). Eventualmente alguns itens do pedido do projeto p1 podem ocorrer no pedido de p2, possivelmente com quantidades distintas. Dizemos que um projeto p1 é subprojeto de p2 se cada componente de p1 é também componente de p2 em quantidade idêntica ou inferior.
4. (lfornecedor) Dado um pedido de compra e a lista de revendedores, descrever a lista de fornecedores para cada componente, com seus respectivos preços, no formato

   [(material1,[rev1, rev2,...]), (material2, [...]), ...]

18.5 Gerência Acadêmica: Considere a gerência acadêmica dos cursos de graduação de uma universidade. As disciplinas cursadas por um aluno são registradas em seu histórico. O registro deve conter o código da disciplina, o ano e o semestre em que foi cursada e a nota obtida. Semestralmente o aluno deve requerer matrícula em novas disciplinas. O pedido de matrícula é realizado através da apresentação das disciplinas desejadas pelo aluno. Um dos critérios para conseguir se matricular em uma disciplina é que o aluno tenha cumprido, com aprovação, os pré-requisitos da disciplina. Um aluno é aprovado em uma disciplina se obtiver média superior ou igual a 5 (cinco). Cada curso possui uma grade curricular que relaciona cada disciplina do curso com seus respectivos pré-requisitos.

Considere as seguintes definições:

Histórico Escolar	Um par contendo o código de matrícula do aluno e a lista de disciplinas cursadas. Cada disciplina cursada é representada por uma tripla com o código da disciplina, o ano e o semestre que ela foi cursada e a nota obtida. Formato: (matrícula, [(disciplina, (ano, semestre), nota), ...]) onde matrícula = (ano, curso, númeroIdentificaçãoIndividual)
Cadastro de Disciplinas	Uma lista contendo, para cada disciplina, um par com o código da disciplina e a quantidade de créditos.
Grade Curricular	Um par com o código do curso e uma lista de pares onde cada par representa uma disciplina (código) e uma lista dos seus pré-requisitos. Formato: (curso, [(disciplina, [disciplina, disciplina,...]), ...])
Pedido de matrícula	Um par com o código do aluno e uma lista contendo o código das disciplinas nas quais o aluno deseja se matricular. Formato: (matrícula, [pedido1, pedido2, ...])
Oferta	Uma lista contendo as turmas a serem ofertadas, com seus horários e limite de vagas. Formato: [(disciplina, turma, quantidade_de_vagas, horário),...] onde horário = [(dia_da semana, hora_inicial, hora_final), ...]
Lista de disciplinas de um curso, apresentadas por período	Um par contendo o código do curso e uma lista de pares, onde cada par representa um período do curso e uma lista das disciplinas do período. Formato: (curso, [(período, [disciplina, disciplina,...])

Histórico Escolar	Um par contendo o código de matrícula do aluno e a lista de disciplinas cursadas. Cada disciplina cursada é representada por uma tripla com o código da disciplina, o ano e o semestre que ela foi cursada e a nota obtida. Formato: (matrícula, [(disciplina, (ano, semestre), nota), ...]) onde matrícula = (ano, curso, registro)
Cadastro de Disciplinas	Uma lista contendo, para cada disciplina, um par com o código da disciplina e a quantidade de créditos.
Grade Curricular	Um par com o código do curso e uma lista de pares onde cada par representa uma disciplina (código) e uma lista dos seus pré-requisitos. Formato: (curso, [(disciplina, [disciplina, disciplina,...]), ...])
Lista de disciplinas de um curso, apresentadas por período	Um par contendo o código do curso e uma lista de pares, onde cada par representa um período do curso e uma lista das disciplinas do período. Formato: (curso, [(período, [disciplina, disciplina,...])

Exercícios: Escreva funções em Hugs para resolver os problemas propostos abaixo, usando os nomes indicados em negrito. A palavra semestre, nestes problemas, representa um elemento do tipo (ano,s), onde s = 1 ou s = 2.

1. **(credse)** Dado um histórico, o cadastro de disciplinas e um semestre, descrever o total de créditos cumpridos no semestre.
2. **(ncred)** Dado um histórico e o cadastro de disciplinas, descrever a quantidade de créditos cumpridos por semestre.
3. **(dncursadas)** Dado um histórico e uma lista com as disciplinas por período, descrever a lista das disciplinas não cursadas ainda pelo aluno, dos períodos já cumpridos. Esta lista deve ter o seguinte formato: [(período,[disciplina,..]),(período,[disciplina,...]),...]
4. **(maxcred)** Dada uma lista de históricos, identificar a lista de alunos que possuem o maior número de créditos cumpridos.
5. **(conterep)** Dado um histórico, contar o número de reprovações.
6. **(jubila)** Verificar se um determinado aluno se encontra em situação de jubilamento (possui três reprovações em uma mesma disciplina).
7. **(abandono)** Verificar se o aluno deixou de cursar disciplinas em algum semestre.
8. **(divida)** Obter a lista das disciplinas que foram cursadas por um dado aluno, nas quais ele ainda não obteve aprovação.
9. **(reprova)** Obter uma lista de reprovações por período, para um dado aluno.
10. **(crend)** O coeficiente de rendimento (CR) de um dado aluno é determinado pela média ponderada das notas obtidas, tendo como peso, o número de créditos das disciplinas. Determine o CR de um dado aluno.
11. **(semprereq)** Obter uma lista das disciplinas de um dado curso, que não possuem pré-requisito.
12. **(cumprereq)** Dada uma disciplina, verificar se o aluno cumpriu seus pré-requisitos.
13. **(dividas)** Obter para um dado aluno, a lista de disciplinas que ele ainda não cursou, com respeito a uma dada grade curricular.
14. **(sugestao)** Obter para um dado aluno, a lista de disciplinas que ele está apto a cursar, com respeito a uma dada grade curricular.
15. **(lcumprereq)** Obter uma lista de pedidos de disciplinas para as quais os pré-requisitos foram cumpridos.
16. **(ldiscsol)** Dada uma lista com os pedidos de todos os alunos, obtenha uma lista das disciplinas solicitadas, sem repetição.
17. **(sol_disciplinas_ord)** Obter a lista de alunos que solicitaram uma dada disciplina, com seus respectivos coeficientes de rendimentos. A lista de alunos deve estar em ordem crescente por coeficiente de rendimento. A lista resultante deve ter o seguinte formato:

    [(aluno1, coeficiente_de_rendimento1),(aluno2, coeficiente_de_rendimento2), ...]

18.6 Agência de Turismo: Uma agência de turismo possui o mapa atualizado de reservas dos hotéis de uma determinada região. Para cada hotel, todas as informações pertinentes aos tipos de quarto oferecidos são registrados no mapa. Este mapa é representado por uma lista de tuplas, seguindo o seguinte formato:

[(**nome_hotel**, [(**tipo_quarto**, **valor_diária**, **situação**), ...]), ...] onde

- **nome_hotel** é do tipo string;
- **tipo_quarto** é uma string;
- **valor_diária** é um número real positivo;
- **situação**: número inteiro positivo que indica a quantidade de quartos livres do tipo indicado.

Para o melhor entendimento da representação da lista de hotéis no hugs, considere o seguinte exemplo:

```
lhoteis = [("peterle", [("simples", 50.0, 5),("duplo", 75.8, 10), ("luxo", 110.0, 2)] ),
           ("ibis", [("simples", 45.0, 3),("duplo", 65.5, 15), ("luxo", 95.0, 3)] ),
           ("novotel", [("simples", 65.6, 10),("duplo", 90.5, 20), ("luxo", 150.0, 10)] )]
```

Exercícios: Considerando o formato desta lista de hotéis apresentada acima, resolva o seguinte problema, descrevendo funções em HUGS.

1. Dados um tipo de quarto e a lista de hotéis, apresente a lista com o nome do hotel ou com os nomes dos hotéis que possuem a oferta mais barata para aquele tipo de quarto.
2. Avalie a função definida no problema 1 para os parâmetros adequados, considerando a lista de hotéis como a dada acima.

18.7 Espetáculos Teatrais: Vários espetáculos estão sendo apresentados em um grande teatro da cidade. Para cada um dos espetáculos, registra-se o mapa de ocupação da platéia, conforme as vendas dos ingressos. A platéia está representada por *m* filas numeradas de *1* a *m*, sendo que cada fila contém *n* cadeiras também numeradas de *1* a *n.* Considere a seguinte representação para os dados:

Lugar na platéia	**(fila, cadeira)**, onde **fila** é representada por um inteiro de *1* a *m* e **cadeira**, por um inteiro de *1* a *n*.
Platéia	Lista de duplas **(lugar, situação)** sendo que a situação é : *1* para indicar lugar ocupado e *0* para indicar lugar vago.
Teatro	Lista de duplas **(espetáculo, platéia)** onde **espetáculo** é representado por um inteiro de *1* a *p*.

Exercícios: Escreva um script em HUGS, com funções que resolvam os problemas abaixo. Nomes para cada uma das funções são sugeridos ao final do enunciado de cada problema.

5. Dada uma platéia pls, descreva a quantidade total de lugares ocupados (totalOcup).
6. Dado um lugar lg e uma platéia pls, verifique se o lugar lg está livre (estaLivre).
7. Dado um lugar lg e uma platéia pls, verifique se existe algum vizinho lateral de lg que está livre (vizinhoLivre).

8. Dada uma fila fl e uma platéia pls, descreva a lista de cadeiras livres da fila fl (cadeirasLivresFila).
9. Dada uma platéia pls, descreva a lista de cadeiras livres para cada fila (lugLivresFila)
10. Dada uma platéia pls, descreva a(s) lista(s) com o maior número de cadeiras livres (filaMaxLivre).
11. Dado um teatro trs e um espetáculo ep, descreva a sua platéia (plateiaEsp).
12. Dado um teatro trs, um espetáculo ep e uma fila fl, descreva a lista de cadeiras livres da fila fl (cadeirasLivresFilaEsp).

19. ENTRADA E SAIDA DE DADOS

19.1 INTRODUÇÃO:

Os programas que apresentamos e propusemos até então, se basearam em um padrão específico de interação entre o usuário e o sistema. O interpretador HUGS está constantemente solicitando uma expressão, que em seguida é avaliada por ele e o resultado é apresentado ao usuário. Vejamos o modelo ilustrado na figura 20.1. A avalição de uma expressão leva em conta as definições disponíveis em bibliotecas e as construídas pelos programadores.

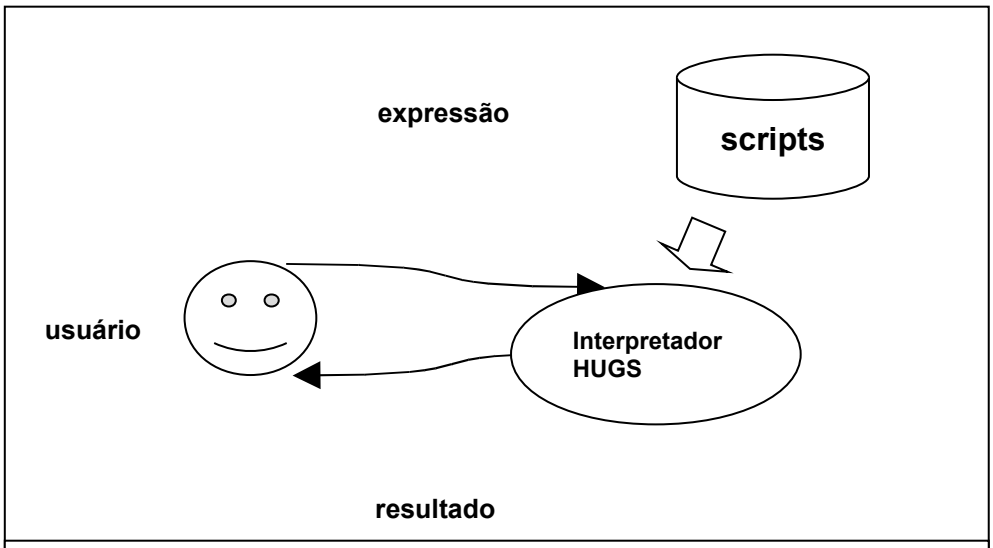


Figura 20.1 – Modelo de Interação usuário-interpretador HUGS

Algumas vezes entretanto este modelo de interação não é adequado tendo em vista que o programa do usuário não pode tomar a decisão de pedir ou não um dado ao usuário. Tudo tem que ser informado antecipadamente.

Exemplo 1: Desejamos fazer um programa que interaja com o usuário para que ele tente descobrir um número oculto. A interação do programa só deve continuar enquanto o jogador não descobrir o número oculto. E veja, não dá para antecipar antes e codificar isto em uma expressão. A Figura 20.2 mostra um esquema da interação necessária. O programa pode começar pedindo um valor ao usuário. Se o usuário fornece um valor idêntico ao valor oculto, o programa exibe a mensagem anunciando que o jogador venceu e encerra o jogo. Caso contrário o programa solicita um novo palpite.

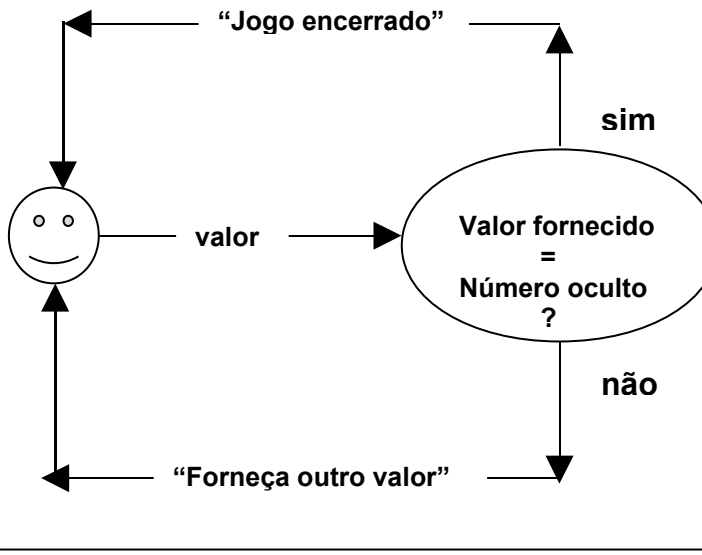


Figura 20.2 – Interações para descobrir um número oculto

Nestes casos, não queremos descrever mapeamentos entre conjunto de valores e sim seqüência de ações. É o contato das funções com o mundo. Ainda podemos usar funções para determinar valores, mas quando precisamos, temos que estabelecer uma seqüência de ações a serem executadas. Agora o que nos interessa é definir programas, através de seqüências de ações.

Vejamos uma solução para o joguinho acima introduzido:

```
jogo1 = do
        putStrLn ">>forneça um número entre 0 e 10"
        valor <- getLine
        if valor == "5"
           then do putStrLn "acertou - parabéns!"
           else do putStrLn "errou - tente outra vez!"
                   jogo1
```

E algumas interações:

```
Main> jogo1
>>forneça um número entre 0 e 10
3
errou - tente outra vez!
>>forneça um número entre 0 e 10
4
errou - tente outra vez!
>>forneça um número entre 0 e 10
2
errou - tente outra vez!
>>forneça um número entre 0 e 10
5
acertou - parabéns!
```

Vamos agora dissecar o programa:

1. Para estabelecer contato com o mundo precisamos de uma ação para leitura e outra para exibição de resultado. Aqui a leitura é realizada através da primitiva **getLine**. Para exibir resultados fazemos uso da primitiva **putStrLn.** Enquanto **getLine** não precisa de um parâmetro, visto que é uma ação que sempre buscará na interação obter um valor, a **putStr** necessita de um valor para ser exibido;
2. O programa **jogo1** não precisa de parâmetro para realizar sua atividade, dado que tudo será solicitado diretamente dentro do corpo do programa, na medida do necessário;
3. Para informar que o que estamos definindo é uma seqüência de ações e não uma descrição funcional, é introduzida a partícula **do**, que pode ser traduzida pelo imperativo **faça!**
4. Para que um valor obtido pelo comando de leitura **getLine**, se torne disponível para uso do programa precisamos internalizá-lo através do uso da primitiva representada pelo símbolo "←";

```
valor ← getLine
```

5. Para indicar que desejamos continuar executando o programa podemos fazer uma chamada a ele mesmo, como o fizemos em:

```
else do putStrLn "errou - tente outra vez!"
        jogo1
```