tutorial de haskell

Engenharia Civil

Universidade Federal do Rio de Janeiro (UFRJ)

45 pag.

tutorial de haskell

http://www.marcosrodrigues6.hpg.ig.com.br/index.htm

Prefácio

A tecnologia dos computadores muda com muita freqüência; os fundamentos, no entanto, permanecem estáticos. A arquitetura de um computador padrão é adaptada de máquinas construídas a meio século. Em programação, idéias modernas como programação orientada a objetos tem décadas de idade para que ter sido adotada no ambiente comercial. Dessa forma, uma linguagem funcional como Haskell é relativamente jovem, mas seu crescimento rápido influencia sua adoção.

> Linguagens funcionais são usadas como **componentes** de sistemas maiores
> Linguagens funcionais são providenciam um **framework** no qual as idéias cruciais da programação moderna estão presentes da forma mais clara possível.

O que é programação funcional?

Programação funcional oferece uma visão de alto nível da programação, tendo com seus usuários uma variedade de recursos que ajudam a criar elegantes e ainda poderosas bibliotecas gerais de funções.

Um exemplo do poder e generalidade da linguagem, é a função *map*, a qual é usada para transformar todo elemento da lista de objetos de uma forma específica. Por exemplo, map pode ser usada para dobrar todos os números em uma sequência numérica ou inverter as cores de cada figura de uma lista de imagens.

A elegância da programação funcional é um conseqüência da forma que as funções são definidas: uma equação é usada para dizer qual o valor da função para uma entrada arbitrária. Um exemplo simples é a função addDouble a qual adiciona dois inteiros e dobra sua soma. Ele é definida como:

```
addDouble x y = 2*(x+y)
```

onde x e y são as entradas e 2*(x+y) é o resultado.

Haskell e Hugs

Haskell começou a ser desenvolvido em 1980 como uma linguagem padrão para programação funcional preguiçosa, e desde então tem sofrido várias atualizações e modificações. Este texto é escrito em Haskell 98, wue consolida o trabalho de Haskell e o qual é suficientemente estável, futuras extensões de resultar no Haskell 2 daqui a alguns anos, mas é esperado implementações que continuem suportando Haskell 98 ainda por um bom tempo.

Por que aprender programação funcional?

Uma linguagem de programação funcional parte de um modelo simples de programação: um valor, o resultado, é computado com base em outras entradas

Porque seu fundamento é simples, uma linguagem funcional tem a visão clara da idéia central na computação moderna, incluindo aabstração (em uma função), abstração de dados (em um tipo abstrado de dados (TAD)), generalidade, polimorfismo e sobrecarga. Tendo isso em vista, as linguagens funcionais provêem uma introdução ideal das técnicas modernas de programação. Por exemplo, Haskell tem uma implementação direta dos tipos de dados como árvores, enquanto outras linguagens são forçadas a a definí-la usando a estrutura de dados de ponteiros.

Haskell não é uma boa "linguagem de ensino"; ela é uma linguagem de programação prática, suportada por ter extensões com interfaces de funções C e programação baseada em componentes, por exemplo. Haskell é também usada em vários projetos do "mundo-real". A Microsoft contratou praticamente toda a equipe que desenvolveu o Haskell para trabalhar na sua plataforma .NET. E existe o Haskell-script, que é uma linguagem alteranativa ao VBA, da própria MS.

Capítulo 1 - Introdução à

programação funcional

Nos últimos cinquenta anos os computadores tem evoluído de unidades enormes, caros, raros e lentos para unidades pequenas, baratos, comuns, rápidas e (relativamente) dependentes. Os primeiros computadores eram máquinas '*stand-alone*' (isoladas), mas hoje eles também podem seguir diferentes regras, sendo organizados em rede, ou sendo integrados em máquinas domésticas como carros e máquinas de lavar, da mesma forma como aparecem em computadores pessoais e PDAs.

A despeito disso, os funtamentos dos computadores tem mudado muito pouco nesse período: o propósito de um computador é manipular informações simbólicas. Esta informação pode representar uma situação simples, como a compra de um ítem em um supermercado, ou uma mais complicada, como a previsão de tempo na Europa.

Como essas tarefas são arquivadas? Nós precisamos escrever uma descrição de como a informação é manipulada. Isto é chamado **programa** e é escrito em uma **linguagem de programação**. Uma linguagem de programação é uma linguagem formal, artificial usada para dar instruções ao computador. Em outras palavras, a linguagem é usada para escrever o **software** que controla o comportamento do **hardware**. Enquanto a estrutura dos computadores tem ficado muito semelhante ao que eram em sua concepção, a forma que eles são programados tem desenvolvido substancialmente. Inicialmente programas são escritos usando instruções que controlavam o hardware diretamente. As linguagens de programação modernas tem trabalhado independente desse problema em outro nível (alto nível), especialmente melhor que as de "baixo nível".

Capítulo 2 - Iniciando a

programação Haskell em Hugs

O capítulo 1 introduziu os fundamentos da programação funcional em Haskell. Vamos agora usar o sistema Hugs para alguma programação prática, e o principal propósito deste capítulo é dar uma introdução ao Hugs.

Iniciando o programa, vamos aprender os módulos básicos do Haskell, nos quais os programas podem ser escritos em múltipls arquivos interdependentes, e que podem usar as funções "embutidas" nas bibliotecas de prelúdio.

2.1 Um primeiro programa em Haskell

Começaremos o capítulo tomano um primeiro programa Haskell ou **script**, que consiste de exemplos numéricos do Capítulo 1. Como foi definido, um script pode conter comentários.

```
{-
##########################################
################
FirstScript.hs
Simon Thompson, Junho 1998
O propósito desse script é:
- ilustrar alguns exemplos com inteiros (Int)
- mostrar o primeiro exemplo de um script
##########################################
##################-}
-- O valor de size é um inteiro (Int), definido como a

-- soma de doze e treze
size :: Int
size = 12+13

-- a função para o quadrado de um inteiro
square :: Int -> Int
square n = n*n

-- A função para o dobro de um inteiro
double :: Int -> Int
double n = 2*n

-- Um exemplo usando double, square e size
example :: Int
```

```
example = double (size - square (2+2))
```

Figura 2.1. Um exemplo de um script tradicional

Clique aqui para baixar o código fonte de FirstScript.hs

Um comentário em um script é uma peça chave de informação para uma pessoa que estuda o código. Ele pode conter uma explanação informal aobre como uma função trabalha, como ele pode ou não pode ser usada, assim por diante.

Existem dois estilos diferentes de scripts em Haskell, que refletem duas diferentes filosofias de programação.

Tradicionalmente, todo código do programa é interpretado com o texto do programa, *exceto* onde é explicitamente indicado que é comentário. Este é o estilo de FirstScript.hs, na Figura 2.1. Scripts deste estilo são armazenados em arquivos com a extensão '.hs'.

Comentários são indicados de duas formas. O símbolo '--' começa um comentário que que ocupará o restante da linha à direita do símbolo. Comentários também podem ser delimitados pelos símbolos '{-' e '-}'. Estes comentários podem ser de tamanho arbitrário, podendo ser de mais de uma linha, bem como englobando outros comentários; então eles são chamados de ***nested comments***.

```
########################################
################
FirstScript.hs
Simon Thompson, Junho 1998
O propósito desse script é:
- ilustrar alguns exemplos com inteiros (Int)
- mostrar o primeiro exemplo de um script
########################################
################
O valor de size é um inteiro (Int), definido como a


soma de doze e treze
> size :: Int
> size = 12+13


a função para o quadrado de um inteiro
> square :: Int -> Int
> square n = n*n


A função para o dobro de um inteiro
> double :: Int -> Int
```

```
> double n = 2*n


Um exemplo usando double, square e size
> example :: Int
> example = double (size - square (2+2))
```

Figura 2.2. Um exemplo de um literate script

Clique aqui para baixar o código fonte de FirstScript.lhs

A alternativa, o **literate** considera que qualquer coisa escrita é um comentário do programa, e o Código do programa precisa ser sinalizado explicitamente da mesma forma. Uma versão literate do script está na Figura 2.2, onde ele pode ser visto em uma linha começando com '>', e é separado do resto do texto por linhas em branco. Literate scrips são armazenados em arquivos de extensão '.lhs'.

As duas formas enfatizam diferentes aspectos da programação. A tradicional preferência para o programa, enquanto o literate enfatiza que existe mais programação que simplesmente criar definições. Projetar decisões precisa ser explicado, condições para o uso de funções a assim por diante precis ser escrito e detalhado - isto é benéfico para ambos, o usuário e o programador e de fatopara nós mesmos se formos olhar para um código escrito no passado, e precisarmos modificá-lo e entendê-lo.

2.2 Usando Hugs

Hugs é uma implementação de Haskell que roda em PCs (sob Windows 9x/NT/2000) e sistemas Unix, incluindo Linux e recentemente em Mac (Mac OS X). Ele é livremente distribuído na Home Page do Haskell

http://www.haskell.org/hugs/

Embora disponível em diversas plataformas, os programas apresentam melhor desempenho na plataforma Unix. Vale lembrar que o Hugs é um interpretador. O compilador Haskell mais usado é o ghc. Mas ainda existe muita pesquisa na área para criar um compilador mais eficiente. Hoje a maioria da equipe que criou o Haskell trabalha para a Microsoft e ajudam a desenvolver a .NET.

Iniciando o Hugs

Para iniciar o Hugs o Unix, digite *hugs* do prompt; para executar Hugs usando um arquivo particular, digite Hugs seguido do nome do arquivo em questão, como em

```
hugs FirstLiterate
```

Em sistemas Windows, Hugs é executado a partir do Menu Iniciar no local designado na hora da instalação; para executar Hugs para um arquivo particular, dê um clique duplo no ícone do arquivo em questão.

Scripts Haskell trazem a extensão .hs ou .lhs (para literate scripts); somente esses arquivos podem ser carregados, e suas extensões podem ser omitidas quando Hugs é chamado ou pelo comando **:load command** dentro do Hugs.

Figura 2.3. Uma sessão Hugs no Windows

Avaliando expressões em Hugs

Como foi dito anteriormente, o interpretador Hugs avalia expressões digitadas no prompt. Entre no Hugs e carregue o programa de uma das formas expostas. Podemos testar no Hugs a avaliação de size, bem como dois exemplos mais complexos:

```
Main>double 32 - square (size - double 3)
-297
Main>double 320 - square (size - double 6)
471
Main>
```

Como pode ser visto no exemplo, nós podemos avaliar expressões que usam as definições no script atual. Neste caso, ele é o FirstLiterate.lhs (ou FirstLiterate.hs).

Uma das vantagens da interface do Hugs é que é fácil experimentar as funções, tentando avaliações diferentes simplesmente digitando as expressões no teclado. Se quisermos avaliar uma expressão complexa, ela pode ser fácil adiconar ao programa, como na definição

```
teste :: Int teste = double 320 - square (size - double 6)
```

Feito isso, precisamos apenas digitar *test* no prompt *Main>*

Comandos Hugs

Comandos hugs começam com dois pontos, ':'. Um resumo dos principais comandos seguem abaixo.

:load *arquivo*	Carrega o arquivo Haskell *arquivo.hs* ou *arquivo.lhs*. A extensão .hs ou .lhs não precisa ser incluída no nome do arquivo.
:reload	Repete o último comando load.
:edit *first.lhs*	Edita o arquivo first.lhs no editor de textos default. Note que a extensão é necessária nesse caso. Veja a seção seguinte para mais informações sobre edição.
:type *exp*	Retorna o tipo da expressão exp. Por exemplo, o resultado de digitar **:type size+2** é *Int.*
:info *name*	Retorna informações sobre *name*.

:find *name*	Abre o editor de textos no arquivo contendo a definição de *name*.
:quit	Sai do sistema.
:?	Retorna uma lista dos comandos Hugs.
!com	Executa o comando Unix ou DOS *com*.

Todos os comandos ':' podem ser abreviados pela sua letra inicial, como **:l arquivo** a assim por diante. Detalhes de outros comandos podem ser encontrados na documentação on-line do Hugs que pode ser lida usando um Web browser, como Netscape, Opera, Konqueror ou Internet Explorer.

Editando scripts

Hugs pode ser conectado a um editor de textos "default", assim comandos como **:edit** e **:find** usam este editor. Isto pode ser determinado pela sua configuração local. O editor de texto default do Unix é o *vi*; nos sistemas Windows o Notepad pode ser usado. Detalhes de como o comando **:set** pode ser usado para alterar o editor default será explicado mais adiante neste tutorial.

Usando o comando **:edit** no Hugs causa uma chamada do editor de textos no arquivo apropriado. Quando o editor é encerrado, o arquivo atualizado é carregado automaticamente. No entanto é mais conveniente deixar o editor executando em uma janela separada e recarregar o arquivo usado:

1. Gravar o arquivo atualizado do editor (sem fechá-lo) e
2. Recarregar o arquivo no Hugs usando **:reload *arquivo***

Dessa forma o editor deixará aberto o arquivo facilitando as modificações.

Daremos agora alguns exercícios introdutórios usando Hugs nos primeiros programas de exemplo.

Uma primeira sessão Hugs

Tarefa 1

Carregue o arquivo FirstLiterate.lhs no Hugs, e avalie os seguintes expressões:

```
square size
square
double (square 2)
$$
square (double 2)
23 - double (3 - 1)
23 - double 3+1
$$ + 34
```

13 `div` 5
13 `mod` 5

Com essa base, você pode explicar o que é a função $$?

Tarefa 2

Use o comando Hugs :type para saber o tipo de cada um desses exemplos, incluindo $$.

Tarefa 3

Qual o efeito de se digitar cada uma das expressões abaixo.

double square
2 double

Tente explicar os resultados que você obteve

2.3 A biblioteca Prelude.hs e outras bibliotecas Haskell

Dizemos no Capítulo 1 que o Haskell tem vários tipos embutidos, como integers e listas e funções sobre esses tipos, incluindo funções aritméticas, funções de manipulação de listas, map e ++. Estas definições estão contidas na **standard prelude**, *prelude.hs*. Quando Haskell é usado, o default é carregar a biblioteca prelude, como pode ser visto na figura 2.3 a linha:

Reading file: "C:\hugs\lib\Prelude.hs";

que precede o processamento do arquivo *FirstLiterate.lhs* no qual Hugs foi chamado.

Como Haskell foi desenvolvidoa mais de uma década, o prelude também evoluiu. Começou pequeno, e com o tempo algumas das funções usadas na biblioteca padrão foram movidas para ele, para dar mais liberdade de inclusão de novas funções, quando necessário.

Além das bibliotecas padrão, a distribuição Hugs contribui com várias bibliotecas incluídas que suportam concorrencia, animações funcionais e assim por diante. Outros sistemas Haskell também contribuem com outras bibliotecas, mas todos os sistemas suportam as bibliotecas padrão.

Para trabalhar com as bibliotecas nós precisamos aprender algo sobre módulos em Haskell, que veremos agora.

2.4 Módulos

Uma parte típica de um software de computador contém milhares de linhas código. Para poder gerenciar esse código, precisamos quebrá-lo em pequenos componentes, que chamamos modulos (modules).

Um **module** tem um nome e contém uma coleção de definições Haskell. Para introduzir um módulo chamado Ant, precisamos começar o programa com a texto:

module Ant where

....

Um módulo também pode importar definições de outros módulos. O módulo Bee importará as definições de Ant incluindo uma estrutura import, assim:

module Bee where

import Ant

...

A estrutura import significa que poderemos usar todas as definições de Ant quando criamos definições em Bee. Por esse comportamento dos módulos, nós adotamos as convenções abaixo:

1. Existe exatamente um módulo por arquivo
2. o arquivo *Blah.hs* ou *Blah.lhs* contém o módulo Blah

O mecanimo de módulos suporta várias bibliotecas discutidas na seção 2.3, mas podemos também incluir codigo escrito por nós mesmos ou por alguém.

O mecanismo de módulos permite controlar quais definições são importantes e também quais são disponíveis ou **exportáveis** por um módulo para uso em outros módulos.

Agora estamos em condições de explicar porque o prompt do Hugs aparece como **Main>** . O prompt mostra o nome do módulo de mais alto nível correntemente carregado no Hugs, e na ausência de um nome para o módulo, ele é chamado de módulo 'Main' (principal).

2.5 Erros e mensagens de erro

Nenhum sistema pode garantir que o que você digita está correto. Hugs não é excessão. Se alguma coisa está errada, em uma mensagem de erro ou em um script, você pode receber uma **mensagem de erro**. Tente digitar

2+(3+4

no prompt do Hugs. Esse é um erro de **sintaxe**.

A expressão falta parênteses: depois do '4', um fecha parênteses está faltando, para fechar o parêntese aberto antes do '3'. A mensagem de erro diz que o que se seque ao '4' é inexperado:

ERROR - Syntax error in expression (unexpected end of input)

De forma similar, digitando **2+(3+4))** resulta na mensagem

ERROR - Syntax error in input (unexpected `)')

Agora tente digitar a seguinte expressão

double square

Esse é um erro de **tipo**, uma vez que double é aplicado à função square, sendo que ela pede um integer

ERROR - Type error in application

***** Expression : double square**

***** Term : square**

***** Type : Int -> Int**
***** Does not match : Int**

A mensagem indica que alguma coisa do tipo Int era experado, mas alguma coisa do tipo Int -> Int foi apresentada no lugar. Nesse caso, double espera algo do tipo Int como argumento, mas square, do tipo Int -> Int foi encontrado no lugar de um inteiro.

Quando recebemos um erro como o descrito acima, precisamos olhar como o **termo**, neste caso *square* do tipo Int -> Int, não confere no **contexto** no qual é usado: o contexto é dado na segunda linha (double square) e o tipo requerido pelo contexto, Int, é dado na última linha.

Erros de tipo nem sempre nem sempre são mensagens bem estruturadas. Digitando **4 double** ou **4 5** receberemos a mensagem

ERROR - Illegal Haskell 98 class constraint in inferred type

***** Expression : fromInt 4 double**
***** Type : Num ((Int -> Int) -> a) => a**

Será explorado posteriormente detalhes sobre essa mensagem. Por hora é suficiente interpretar esse erro como **Erro de tipo**.

Tente digitar a expressão

4 `div` (3*2-6)

Não existe divisão por zero. Sendo assim recebemos a mensagem

Program error: {primQrmInteger 4 0}

indicando que a divisão de 4 por zero ocorreu.

Capítulo 3 - Tipos básicos e

definições

Abrangemos até agora a base da programação funcional e mostramos como escrever simples programas em Haskell. Este capítulo cobre os mais importantes **tipos básicos** do Haskell e também mostra como escrever definições de funções que têm multiplos **casos** para cobrir situações alternativas. Concluímos olhando alguns detalhes da **sintaxe** do Haskell.

Haskell contém uma variedade de tipos numéricos. Nós já vimos o uso do tipo *Int*; veremos agora este e também o tipo *Float* do **ponto-flutuante** para números fracionários.

Frequentemente em programação nós queremos criar uma opção de valores, se atender ou não uma determinada **condição**. Algumas condições poderiam ser se um número é maior que outro; ou ainda se dois valores são iguais, e assim por diante. O resultado destes testes - `True` se a condição for verdadeira e `False` se for falsa - são chamados de valores **Booleanos**, depois do século XIX por causa do lógico George Boole, e elas formam o tipo **Bool** no Haskell. Este capítulo abrange os Booleanos, e como eles são usados para fornecer escolhas em definição de funções por meiode **quards**.

Finalmente, olhamos o tipo caracter - letras individuais, digitos, espaço e assim por diante - que abrange o tipo **Char** no Haskell.

O capítulo provê um material de referência para os tipos básicos; um leitor pode pular o tratamento de **Float** e precisa detalhar sobre **Char**, retornando a esse capítulo quando necessário.

Cada sessão contém exemplos de funções, e o exercício trata alguns desses. Mais adiante, este capítulo tem os funtamentos dos quais veremos uma variedade de formas de programação podendo planejar e escrever os tópicos deste capítulo.

3.1 Os booleanos: Bool

Os valores booleanos True e False representam o resultado de testes, os quais podem, por instâncias, comparar dois números para igualdade ou verificar se o primeiro é menor que o segundo. O tipo booleano em Haskell é chamado Bool. Os operadores booleanos oferecidos pela linguagem são:

&&	e
||	ou
not	não

Como Bool contém somente dois valores, nós podemos definir o propósito dos operadores Booleanos por **tabelas verdade** que mostram o resultado da aplicação do operador a cada possível combinação de argumentos. Para a instância, a terceira linha da primeira tabela diz que o valor de False && True é False e que o valor de False || True é True.

t1	t2	t1 && t2	t1 || t2
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

t1	not t1
T	F
F	T

Booleanos podem ser os argumentos ou o resultado das funções. "Ou exclusivo" é a função que retorna True exatamente quando um mas não ambos os argumentos tem o valor True; ela é como o "ou" de um menu de restaurante: você pode escolher um prato, mas não ambos!

```
exOr :: Bool -> Bool -> Bool
exOr x y = (x || y) && not (x && y)
```

Podemos ver na figura abaixo uma representação da função, usando retângulos para funções e linhas para valores, como visto no Capítulo 1 . Setas entrando em uma função representam os argumentos e setas saindo representam o resultas.

Valores booleanos podem também ser comparados por igualdade e diferença usando os operadore == e /=, onde ambos são do tipo:

```
Bool -> Bool -> Bool
```

Note que /= é a mesma função que exOr, uma vez que ambos retornam o resultado True quando exatamente um dos argumentos é True.

Literais e definições

Expressões como True e False, e também numeros como "2", são chamados de **literais**. Estes são valores literais que não precisam ser avaliados; o resultado da avaliação de um literal é o próprio literal.

Podemos usar os literais True e False como argumentos, e nós mesmos definir uma função not:

```
mynot :: Bool -> Bool

mynot True = False
mynot False = True
```

Podemos também usar uma combinação de literais e variáveis no lado esquerdo da equação para definir exOr:

```
exOr True x = not x
exOr False x = x
```

Até aqui vimos uma definição que usa duas equações: a primeira aplica toda hora o primeiro argumento para exOr como True e a segunda quando o argumento é False.

Definições que usam True e False no lado esquerdo da equação são frequentemente mais legíveis que definições nas quais somente temos variáveis no lado esquerdo. Isto é um um exemplo simples do mecanismo **pattern matching** no Haskell, o qual será examinado com mais detalhes mais adiante.

Exercícios

1. Crie uma outra versão para a definição de 'exclusive or' que trabalha da seguinte forma: "exclusive or de x e y será True se x é true e y é false, ou vice versa".
2. Crie um digrama de retângulos e linha para sua resposta acima.
3. Usando literais no lado esquerdo da função, podemos criar tabelas verdade para uma função em sua definição Haskell. Complete a seguinte definição neste estilo.

   exOr True True = ...
   exOr True False =
4. Dê duas definições diferentes para a funsção nAnd

   nAnd :: Bool -> Bool -> Bool

   que retorna True exceto quando ambos os argumentos são True. Crie um diagrama ilustrando uma de suas funções.
5. Descreva linha por linha o cálculo de

   nAnd True True

   nAnd True False

   Para cada uma de suas definições de nAnd do exercício anterior

3.2 Os Inteiros: Int

O tipo Haskell **Int** contém o conjunto dos inteiros. Os inteiros são o conjunto de números usados para contagem; eles são escritos da seguinte forma:

0
45
-3452
2147483647

O tipo Int representa os inteiros em uma quantidade fixa de espaço, e somente pode ser representado uma faixa finita de inteiros. O valor maxBount retorna o maior valor do tipo, que sucede 2147483647. Para a maioria dos cálculos inteiros é um número de tamanho fixo são alternaveis, mas se um número maior for requerido, podemos usar o tipo Integer, que pode representar números de qualquer tamanho.

Faremos aritmética em inteiros usando os seguintes operadores e funções; as operações que discutimos aqui também são aplicáveis ao tipo Integer.

+	A soma de dois números
*	O produto de dois inteiros
-	A diferença de dois inteiros, quando infixa: a-b; o inteiro de sinal oposto quando prefixa: -a.
^	Radiciação; 2^3 é 8
div	Quociente da divisão numérica; por exemplo, **div 14 3 é 4.** Isto pode se esrito 14 `div` 3.
mod	O resto da divisão inteira; para o exemplo mov 14 3 (ou 14 `mod` 3) é 2.
abs	O valor absoluto de um inteiro; remove o sinal.
negate	A função altera o sinal de um inteiro.

Note que `mod` cercado por **crases** é escrito entre os dois argumentos, esta é a versão **infixa** da função mod. Qualquer função pode ser escrita da forma infixa.

Nota: literais negativos
Um erro comum ocorre com literais negativos. Por exemplo, o número menos 12 é escrito como -12, mas o prefixo '-' cope gerar confusão com o operador infixo para subtrair um numero de outro e pode gerar uma mensagem de erro confusa e imprevista. Por exemplo, a aplicação **negate -34** é interpretada como 'negate menos 34' e gera no Hugs a mensagem: ERROR - Unresolved overloading *** Type : (Num a, Num (a -> a)) => a -> a *** Expression : negate - 34

Desse ponto em diante, usaremos o termo **números naturais** para os inteiros não negativos: 0, 1, 2, ...

Operadores Relacionais

Existem relações de igualdade e ordenação sobre números inteiros, como existe sobre todos os *tipos* básicos. Estas funções recebem dois inteiros como entrada e retornam um Bool, que pode ser *True* ou *False*. As relações são:

>	maior que (e não igual a)
>=	maior que ou igual a
==	igual a
/=	não igual a
<=	menor que ou igual a
<	menor que (e não igual a)

Um exemplo simples usando essas definições é a função que teste se três Ints são iguais.

```
threeEqual :: Int -> Int -> Int -> Bool
threeEqual m n p = (m==n) && (n==p)
```

Exercícios

1. Explique o efeito da função definida abaixo:
   mystery :: Int -> Int -> Int -> Bool
   mystery m n p = not ((m==n) && (n==p))

   Dica: se você encontrar dificuldades para responder essa pergunta, tente ver o que a função faz com algumas entradas de exemplo.

2. Defina uma função
   threeDifferent :: Int -> Int -> Int -> Bool
   que retorna True somente quando todos os três números m, n e p forem diferentes.
   Qual é a o retorno da sua função para threeDifferent 3 4 3? Explique porquê ele retornou essa resposta.

3. Esta questão é sobre a função

   fourEqual :: Int -> Int -> Int -> Int -> Bool

   Que retorna o valor True somente se todas as quatro argumentos forem iguais. Dê uma definição de fourEqual modelado como a definição de threeEqual acima. Agora dê uma definição de fourEqual que usa a função threeEqual na definição. Compare suas duas respostas.

4. Descreva linha por linha o cálculo de

   threeEqual (2+3) 5 (11 `div` 2)

   mystery (2+4) 5 (11 `div` 2)

   threeDifferent (2+4) 5 (11 `div` 2)

   fourEqual (2+3) 5 (11 `div` 2) (21 `mod` 11)

3.3 Sobreposição (overloading)

Ambos, inteiros e booleanos, podem ser comparados por igualdades, e alguns símbolos == são usados para ambas operações, embora sejam diferentes. Realmente, == será usado para igualdade sobre qualquer tipo t para os quais é possível definir uma operação de igualdade. Isto significa que (==) tem o tipo
Int -> Int -> Bool
Bool -> Bool -> Bool

e realmente t -> t -> Bool se o tipo t aceita igualdade

Usando o mesmo símbolo ou nome para operações diferentes é chamado **overloading**. Um número de símbolos em Haskell são sobrepostos, e veremos mais adiante como sobreposição e manipulação de tipos de sistema em Haskell, e também como usuários podem definir suas próprias sobreposições para nomes ou operadores.

3.4 Guards

Agora vamos explorar como condições, ou **guards** são usadas para dar alternativas em definições de funções. Um guard é uma expressão booleana, e essa expressão é usada para expressar vários casos na definição de uma função.

Mostraremos um exemplo nesta sessão de funções que comparam inteiros por tamanho, e começaremos olhando o exemplo da função que retorna o valor máximo de dois inteiros. Quando os dois números são iguais, então retorna o valor comum como máximo.

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

Como lemos uma definição como essa, que aparece no Haskell em prelude.hs?

Em geral, se o primeiro guard (nesse exemplo, x >= y) é *True*, então o valor correspondente é o resultado (x neste caso). Por outro lado, se o primeiro guard é *False*, então olharemos o segundo caso, e assim por diante. Um guard otherwise é o último dos argumentos e sempre retorna True, que neste caso é para y > x.

Um exemplo de múltiplos guards é uma definição de máximo de três entradas.

```
maxThree :: Int -> Int -> Int -> Int
maxThree x y z
| x >= y && x >= z = x
| y >= z = y
| otherwise = z
```

Como essa definição trabalha? O primero guard

```
x >= y && x >= z
```

testa quando x é o máximo das três entradas; se for True, o resultado é x. Se o guard falha, então x não é o máximo, o que resta escolher entre y e z. O segundo guard é

```
y >= z
```

Se ele for verdadeiro, o resultado é y; senão o resultado é z. Nós voltaremos ao exemplo de MaxThree na próxima sessão.

Primeiro demos a forma geral para definições simples no Capítulo 1. Podemos agora estendê-la a uma forma geral para definições com quards. Note que *otherwise* não é obrigatório.

Figura 3.1. A forma geral para a definição de função com guards

Nós também vimos no Capítulo 1 que podemos escrever linha por linha **cálculos** para os valores das expressões. Como os guards adequam a esse modelo? Quando aplicamos uma função com seus argumentos, nós precisamos saber em quais casos ela se aplica, e pra isso precisamos avaliar os guards até encontrar um guard que seja *True*; uma vez encontrado, podemos avaliar o resultado correspondente. Tomando com o exemplo maxThree, temos dois exemplos os quais mostramos as avaliações de guards em linhas começando por "??".

```
maxThree 4 3 2
?? >= && 4 >= 2
?? True && True
?? True
4
```

Neste exemplo o primeiro guard testa 4 >= 3 && 4 >= 2, que resulta True e assim, o resultado é correspondente ao valor, 4. No segundo exemplo, nós avaliamos mais de um guard.

```
maxThree 6 (4+3) 5
?? 6>=(4+3) && 6>=5
?? 6>=7 && 6>=5
?? False && True
?? False
?? 7 >=5
?? True
7
```

Neste exemplo, nós inicialmente avaliamos o primeiro guard, 6>=(4+3) && 6>=5, que resulta em *False*; depois, avaliamos o segundo guard, 7>=5, que resulta em *True*, a finalmente o resultado é 7.

Uma vez que temos calculado o valor do segundo argumento, (4+3), não recalculamos este valor quando seguimos em diante. Isto não é só um truque de nossa parte; o sistema Hugs somente irá avaliar um argumento como (4+3) uma vez, mantendo esse valor no caso de precisar novamente, ao invés de recalculá-lo. Este é um aspecto da avaliação preguiçosa, que veremos mais adiante.

Expressões condicionais

Guards são condições que distinguem entre casos diferentes em definições de funções. Podemos também escrever expressões condicionais gerais por meio da construção **if...then...else** do Haskell. O valor de

```
if condition then m else n
```

é m se a condição é *True* e é n se a condição for *False*, assim a expressão **if False then 3 else 4 retorna o valor 4,** e no geral

if x >= y then x else y

será o máximo de x e y. Isto mostra qe podemos escrever max em uma forma diferente:

max :: Int -> Int -> Int
max x y
= if x >= y then x else y

Tende-se a usar muito a forma guard, mas veremos nos exemplos onde o uso de if... then ... else .... é mais natural

Nota: redefinindo funções de prelude
A função max é definida no prelúdio, Prelude.hs, e se uma definição max :: Int -> Int -> Int aparecem em um script maxDef.hs então esta definição terá um conflito com a definição do prelúdio, gerando a mensagem de erro no Hugs ERROR "maxDef.hs" (line 3): Definition of variable "max" clashes with import Para redefinir a função preludio max e min, insira a linha import Prelude hiding (max,min) que sobrescreve o import usuao de prelude incluído no topo do arquivo maxDef.hs, depois desta estrutura. Muitas das funções definidas neste texto são de fato incluídas em prelude, e esta técnica precisa ser usada sempre que você precisar para redefinir uma delas.

Exercícios

1. Dê o resultado dos cálculos abaixo
   max (3 -2) (3*8)
   maxThree (4+5) (2*6) (100 `div` 7)
2. Dê a definição das funções
   min :: Int -> Int -> Int
   minThree :: Int -> Int -> Int

3.5 Os caracteres: Char

Pessoas e computadores comunicam usando o teclado como entrada e o monitor como saída, que são baseados em sequências de **caracteres**, como letras, dígitos e caracteres "especiais" como

espaço, tabulação, nova linha, fim de arquivo. Haskell contém um tipo interno para caracteres, chamado *Char*

Caracteres literais são escritos entre aspas simples, assim 'd' é a representação Haskell do caracter d. Similarmente, '3' é o caracter três. Alguns caracteres especiais são representados como os abaixo:

tab	'\t'
newline	'\n'
barra invertida (backslash)	'\\'
aspas simples (')	'\''
aspas dupla (")	'\"'

Existe uma codificação padrão para representar caracteres por inteiros, chamada de codificação ASCII. As letras maíusculas de 'A' a 'Z' tem a seqüência de código de 65 a 90, e as letras minúsculas de 'a' a 'z' código 97 a 122. O caracter com o código 34, por exemplo, pode ser escrito como '\34', e '9' e '\97' têm o mesmo siginifado. ASCII foi recentimente extendida para o padrão Unicode, que contém caracteres de outras fontes além do Inglês.

Existem funções de conversão entre caracteres e seus códigos numéricos que convertem um inteiro em caracter, e vice versa.

```
ord :: Char -> Int
chr :: Int -> Char
```

As funções de codificação podem ser usadas definindo funções sobre Char. Para converter uma letra minúscula para uma maíuscula, um deslocamento precisa ser adicionado ao código:

```
offset :: Int
offset = ord 'A' - ord 'a'

toUpper :: Char -> Char
toUpper ch = chr (ord ch + offset)
```

Note que o offset é chamado, em lugar de aparer como uma parte de toUpper, como em

```
toUpper ch = chr (ord ch + (ord 'A' - ord 'a'))
```

Isto é uma prática padrão, criando ambos os programas é mais fácil para leitura e para modificá-lo. Para mudar o valor do offset, nós precisamos alterar a definição de offset, no lugar de alterar a função (ou funções) que o usam.

Caracteres podem ser comparados usando a ordanação de seus códigos. Assim, desde os dígitos 0 a 9 ocupam um bloco de códigos adjacentes 48 a 57, podemos verificar se um caractere é um dígito assim:

```
isDigit :: Char -> Bool
isDigit ch = ('0' <= ch) && (ch <= '9')
```

O padrão prelude contém um número de funções de conversão, como toUpper, e discrimina funções como isDigit; detalhes podem ser encontrados no arquivo *Prelude.hs*. Outras funções muito usadas sobre Char podem ser encontradas na biblioteca Char.hs.

Nota: caracteres e nomes
É fácil confundir a e 'a'. Para resumir a diferença, a é o nome ou uma variável, que se definida pode ter qualquer valor,

Exercícios

1 Defina uma função para converter letras minúsculas para maiúsculas e não altera caracteres que não são letras minúsculas.
2 Dê a definição da função
charToNum :: Char -> Int
que converte um dígito como '8' para seu valor, 8. O valor de caracteres que não sejam dígitos deve ser 0.

3.5 Os números de ponto flutuante: Float

Na seção 3.2, vimos o tipo Haskell Int para inteiros.Nos cálculos, nós também usamos números com partes fracionais, que são representados no Haskell por números de ***ponto flutuante*** *sendo criados com o tipo Float. Nós não usaremos Float extensamente nas próximas seções, e esta seção pode ser omitida numa primeira leitura e usada como material de referência para ser consultado quando necessário.*

Internamente ao sistema Haskell, existe uma quantidade fixa de espaço alocado para cada Float. Por isso, nem todas as frações podem ser representadas por ponto flutuante, e aritimética sobre elas pode não ser sempre exata. É possível o uso tipo de ponto flutuante de dupla precisão Double para grande precisão, ou para frações de precisão completas de Integer existe o tipo Rational. Mas como esse é um tutorial, vamos nos restringir aos tipos Int e Float, mas nós examinaremos os tipos numéricos momentaneamente no Capítulo 12.

Números de ponto flutuante literalmente em Haskell podem ser escritos por numerais decimais, como em

0.31426
-23.12
567.347
4523.0

Os números são chamados de ponto flutuante porque a posição do ponto decimal não é a mesma para todos os Floats; dependendo de

um número particular, mais de um espaço pode ser usado para armazenar inteiros ou a parte fracional.

Haskell também permite literais numéricos de ponto flutuante em ***notação científica****. Esta, segue a forma abaixo, onde seus valores são dados pela coluna no lado direito da tabela.*

231.61e7	231.61 x 10^7=2.316.100.000
231.61e-2	231.61 x 10^{-2}=2,3161
-3.412e03	-3.412 x 10^3=-3412

Esta representação é mais conveniente que os numerais decimais acima números muito grandes ou muitos pequenos. Considre o número 2.1^{444}*. Ele precisará de centenas de dígitos antes do ponto decimal, e não será possível uma notação decimal já que esta tem um tamanho limitado (usualmente 20 dígitos). Na notação científica, escreveremos como 1.162433e+143;*

Haskell fornece uma faixa de operadores e funções sobre Float no prelúdio padrão. A tabela da Figura 3.2 mostra seu nome, tipo e uma breve descrição do seu comportamento. Estão incluídas:

- *operações matemáticas padrão: raiz quadrada, exponencial, funções logaríticas e trigonométricas.*
- *funções para conversão de inteiro para ponto flutuante: fromInt, e vice versa: ceiling, floor e round*

+ - *	Float -> Float -> Float	Soma, subtração e multiplicação
/	Float -> Float -> Float	Divisão fracionária
^	Float -> Int -> Float	Exponenciação x^n = x^n para um número natural n
**	Float -> Float -> Float	Exponenciação x**y = x^y
==,/=,<,>,<=,>=	Float -> Float -> Bool	Operadores de igualdade e ordenação
abs	Float -> Float	Valor absoluto
acos,asin,atan	Float -> Float	O inverso de cosseno, seno e tangente
ceiling,floor, round	Float -> Int	Converte uma fração para um inteiro, arredondando para cima, para baixo, ou para o inteiro mais próximo
cos,sin,tan	Float -> Float	Cosseno, seno e tangente
exp	Float -> Float	Potência de e
fromInt	Int -> Float	Converte um Int para um Float
log	Float -> Float	Logaritmo na base e

logBase	Float -> Float -> Float	Logaritmo em uma base arbitrária, fornecido no primeiro argumento
negate	Float -> Float	Muda o sinal de um número
pi	Float	A constante pi
signum	Float -> Float	1.0,0.0,-1.0 de acordo com o número se for positivo, zero, ou negativo.
sqrt	Float -> Float	Raiz quadrada (positivo)

Figura 3.2. *Funções e operações de ponto flutuante*

Haskell pode ser usado como uma calculadora numérica. Tente digitar a expressão seguinte no prompt do Hugs:

*sin (pi/4)*sqrt 2*

Sobreposição (overload) de funções e literais

Em Haskell, os números 4 e 2 podem ser ambos Int e Float. Elas são sobrepostas, como discutido na Seção 3.3. Isto também é verdade para algumas funções numéricas; adição para instâncias tem ambos os tipos

Int -> Int -> Int
Float -> Float -> Float

e o operador relacional == , assim por diante estejam disponíveis para todos os tipos básicos. Exploraremos esta idéia de overload mais detalhadamente quando discutirmos classes de tipo.

Nota: convertendo inteiros para números de ponto flutuante
Embora os literals sejam sobrecarregados, não há nenhuma conversão automática de Int para Float. Em geral, se adicionarmos uma quantidade inteira, como *floor 5.6*, a um float, como 6.7, receberemos uma mensagem de erro se digitarmos (floor 5.6) + 6.7 Uma vez tentamos adicionar quantidades de dois tipos diferentes. Temos que converter o Int para um Float para realizar a adição, assim: fromInt (floor 5.6) + 6.7 Onde fromInt transforma um Int para o Float correspondente.

Exercícios

1. *Escreva uma função que retorna a média de três inteiros*
   averageThree :: Int -> Int -> Int -> Float
   Usando esta função, defina a função

   howManyAboveAverage :: Int -> Int -> Int -> Int

Que retorna quantos entradas são maiores que a média dos valores. O restante da questão ajuda na solução da equação quadrática

$a*X^2+b*X+c=0.0$

Onde a,b e c são números reais. A equação retorna

- *duas raízes reais, se* $b^2 > 4.0*a*c$;
- *uma raiz real, se* $b^2 == 4.0*a*c$;
- *nenhuma raiz real, se* $b^2 < 4.0*a*c$;

Ele assume que a é diferente de zero - este caso é chamado ***non-degenerate****. Em casos degenerate, existem três subcaos:*

- *uma raiz real, se b /= 0.0;*
- *nenhuma raiz real, se b == 0.0 e c /= 0.0;*
- *todos os números reais, se b == 0.0 e c == 0.0;*

2. *Escreva uma função*

   numberNDroots :: Float -> Float -> Float -> Int

   que recebe os coeficientes de uma equação quadrática, a, b e c, e retorna quantas raízes a equação possui. Você pode assumir que a equação é non-degenerate.

3. *Usando sua resposta da última questão, escreva uma função*

   numberRoots :: Float -> Float -> Float -> Int

   que recebe os coeficientes da equação quadrática, a, b e c, e retorna quantas raízes a equação possui. No caso da equação ter todos os números como raíz, você pode retornar 3 como resultado.

4. *A fórmula para as raízes de uma equação quadrática é*

   Escreva definições para as funções

   smallerRoot,largerRoot :: Float -> Float -> Float -> Float

   que retorna a menor e maior raiz real da equação quadrática. No caso da equação não possuir raiz real ou ter todos os valores como raízes, retorne zero como resultado de cada uma das funções.

3.7 Sintaxe

A sintaxe de uma linguagem descreve todas as propriedades dos programas formados. Esta sessão mostra vários aspectos de sintaxe do Haskell, enfatizando principalmente aqueles que podem parecer incomuns à primeira vista.

Definições e layout

Um script contém uma série de definições, uma após outra. Como ele esclarece onde uma definição termina e a outra começa? Em português, o final de uma sentença é sinalizado por um ponto final "." . Em Haskell, o ***layout*** *do programa é usado para mostrar onde uma definição termina e a próxima começa.*

Formalmente, uma definição é terminada pela primeira parte de texto que se encontra na mesma identação ou à esquerda do começo da definição.

Quando escrevemos uma definição, o primeiro caracter abre uma caixa que contém a definição, assim

O que quer que seja digitado na caixa faz parte da definição

...até encontrar algo na linha ou à esquerda da linha. Isto fecha a caixa, assim

Quando estiver escrevendo uma seqüência de definições, é aconselhável que todas estejam no mesmo nível de identação. Em nossos scripts escreveremos sempre as definições top-level que começam no lado esquerdo da página, e em literate scripts identaremos o começo de cada definição por uma única tabulação.

Esta regra de layout é chamada de ***regra de escanteio (offside rule)*** *porque é baseada na idéia de escanteio no futebol. A regra também trabalha para equações condicionais como max e maxThree que consistem de mais de uma cláusula.*

Isto é, de fato, um mecanismo no Haskell para ter um final explícito da parte de uma definição, como um '.' faz em Português: o símbolo 'fim' no Haskell é o ';'. Podemos, para a instância, usar ';' se for escrever mais de uma definição em uma única linha, assim:

answer = 42; facSix = 720

Nota: erros de layout
É possível receber mensagens de erro envolvendo ';' mesmo sem usá-lo. Se quebrarmos a regra de escanteio assim: funny x = x + 1 recebemos uma mensagem de erro como ERROR ... : Syntax error in expression (unespected ';') Uma vez que internamente o sistema coloca um ';' antes do 1

para marcar o final da definição, que certamente coloca o ponto e vírgula em um ponto inexperado.

Layout recomendado

A regra de escanteio permite vários estilos diferentes de layout. Neste tutorial, para definições de qualquer tamanho vamos usar a forma

para uma ***equação condicional*** *construída a partir de um número de cláusulas. Neste layout, cada* ***cláusula*** *começa com uma nova linha, e o guard e seus resultados são alinhados. Note também que por convenção neste texto, sempre especificamos um tipo de função sendo definida.*

Se qualquer expressão e_i pi giard g_i for particularmente grande, então o guard pode aparecer em uma linha (ou linhas) de seu próprio alinhamento, como em

Nomes em Haskell

Até agora vimos uma grande variedade de uso de nomes em definições de expressões. Em uma definição como

```
addTwo :: Int -> Int -> Int
addTwo first second = first+second
```

os nomes ou ***identificadores*** *Int, addTwo, first e second são usados para nomear um tipo, uma função ou duas variáveis. Identificadores em Haskell precisam começar com uma letra - minúscula ou maiúscula - seguida por uma seqüência opcional de letras, dígitos, underscores '_' e aspas simples.*

Os nomes usados em definições de valores precisam começar com uma letra minúscula, como variáveis e variáveis de tipo, que serão introduzidas mais adiante. Por outro lado, letras maiúsculas são usadas para iniciar nomes de tipos, como Int;contrutores, como True e False;nome de módulos e também nome de classes de tipos, as quais nós encontraremos abaixo.

Uma tentativa de dar a uma função um nome que começa com letra maíuscula, como

```
Fun x = x + 1
```

resulta em uma mensagem de erro 'Undefined constructor function "Fun" '.

Existem algumas restrições sobre como identificadores podem ser escolhidos. Existe uma pequena coleção de ***palavras reservadas*** *que não podem ser usadas; elas são*

case class data default deriving do else if import in infix infixl infixr instance let module newtype of then type where

Os identificadores especiais, como qualified, e hiding tem funções especiais em certos contextos mas pode ser usado como identificadores ordinarios.

Por convenção, quanto damos nomes de mais de uma palavra, nós capitalizamos as primeiras letras da segunda palavra e subsequentes, como em 'maxThree'.

Um mesmo identificador pode ser usado para dar nome a uma função e uma variável ao mesmo tempo, ou para um tipo e um contrutor de tipo; nós recomendamos fortemente que você não faça isso, pois pode gerar confusão.

*Se quiser **redefinir** um nome que já está definido no preludio ou em uma das bibliotecas, é possível **esconder** o nome ao importar; detalhes de como isso pode ser feito, foi visto anteriormente.*

Haskell foi construído nos padrões de caracter Unicode, que permite símbolos e fontes além do padrão ASCII. Estes símbolos podem ser usados em identificadores e como em Caracteres Unicode - que são descritos em ums seqüência de 16 bits - podem ser lidos to Haskell na forma \uhhhh onde cada h é um dígito hexadecimal (4 bits). No tutorial nós usaremos exclusivamente o subconjunto ASCII do Unicode.

***Obs.:**até o momento, o Hugs não oferece suporte a Unicode.*

Operadores

*A linguagem Haskell contém vários operadores, como +, ++ e assim por diante. Operadores são funções **infixas**, que são escritas entre seus argumentos, ao invés de antes deles, como no caso de funções ordinárias.*

Em princípio é possível escrever todas as aplicações de um operador com parênteses, assim:

```
(((4+8)*3)+2)
```

mas expressões maiores ficariam difíceis de ler. Com duas propriedades extra de operadores, é possível escrever expressões sem necessitar de delimitá-las por parênteses.

Associatividade

*Se desejarmos adicionar três números: 4, 8 e 99 tanto podemos escrever 4+(8+99) quanto (4+8)+99. O resultado é o mesmo, qualquer que seja a forma que foi escrita, uma propriedade que chamamos **associatividade** da adição. Por isso podemos escrever*

```
4+8+99
```

para a soma, sem gerar ambiguidade. Nem todos os operadores são associativos, no entanto; o que acontece quando escrevemos

4-2-1

para a instância? As duas diferentes formas de inserir parênteses são

(4-2)-1 = 2 - 1 = 1	associatividade a esquerda
4-(2-1) = 4 - 1 = 3	associatividade a direita

Em Haskell, cada operador não associativo é classificado como associativo a esquerda ou a direita. Se for associativo a esquerda, qualquer dupla ocorrência do operador será parentizado a esquerda; se for associativo a direita, será à direita. A escolha é arbitrária, mas segue o costume tanto quanto possível, e em particular, '-' é para ser associado a esquerda.

Precedência

A forma com que os operadores são associados permite resolver expressões como

2^3^2

onde o mesmo operador ocorre duas vezes, mas o que é feito quando dois operadores diferentes ocorrem, como na expressão abaixo?

*2+3*4*
*3^4*2*

Para esse propósito a ***precedência*** *do operador é necessária para compará-los. * têm precedência 7 enquanto + tem 6, assim em 2+3*4 o três é multiplicado por 4 e depois somado com 2, como*

*2+3*4 = 2 + (3*4)*

*De forma similar, ^ tem precedência 8 sendo avaliado primeiro que *, então*

*3^4*2 = (3^4)*2*

Uma tabela completa de associatividade e precedência dos operadores predefinidos em Haskell é dada mais adiante. Na seção 'faça-você-mesmo seus operadores' abaixo, será discutido como operadores são definidos em scipts e também como sua associatividade e precedência podem ser definidos ou alterados por declarações.

Nota: aplicação de função
A precedência mais alta é a aplicação de função, que é dada escrevendo o nome da função e em frente os seus argumentos: f v1 v2 ... v_n. Esta é a precedência mais alta que qualquer outro operador, assim, f n+1 é interpretado como f n somado com 1, (f n) + 1, se quisermos f aplicado a n+1, f (n+1). Se estiver em dúvida, ele é sensivel a parentizar cada cada arguemnto para uma aplicação de função.

Operadores e funções

Operadores infixos podem ser escritos antes dos argumentos, delimitando o operador em parentheses. Temos, conseqüentemente, por exemplo,

(+) :: Int -> Int -> Int

assim

(+) 2 3 = 2 + 3

Esta conversão é necessária depois quando criarmos funções como argumento de outras funções. Podemos também converter funções em operadores delimitando a função por crases. Temos assim, usando a definição de máximo anterior,

2 `max` 3 = max 2 3

Esta notação pode criar expressões envolvendo ***binario*** *ou funções de dois argumentos substancialmente mais legíveis.*

A precedência e a associatividade dos operadores pode ser controlada.

"Faça você mesmo" seus operadores

A linguagem Haskell permite definir operadores infixos diretamente em exatamente da mesma forma das funções. Nomes de operadores são dados por símbolos de operadores inclusos nos símbolos ASCII.

*! # $ % & * + . / < = > ? \ : - ~*

junto com os símbolos Unicode. Um nome de operador pode não começar com dois pontos.

Para definir o operador &&& como uma funçã o mínimo para inteiros, escrevemos:

(&&&) :: Int -> Int -> Int

x &&& y

```
  | x > y
  | otherwise = x
```

A associatividade e precedência de operadores pode ser especificada.

Exercícios

Reescreva suas soluções dos exercícios anteriores para o Layout recomendado.

Dadas as definições

```
funny x = x+x
 peculiar y = y
```

Explique o que acontece quando retiramos o espaço antes de peculiar

Resumo

Neste capítulo introduzimos os tipos básicos Int, Float, Char e Bool junto com várias funções internas. Vimos também como expressões booleanas - chamadas guards - permitem definições com vários casos, e exemplificamos por funções que retornam o maximo de dois argumentos inteiros. Esta definição contém dois casos, um que aplica quando o primeiro argumento é maior e o outro quando o segundo é maior.

Finalmente, vimos como o layout de um programa Haskell é importante - o fim de uma definição é implicitamente dado pela primeira parte do texto de do outro programa que começa pela definição; vimos também um overview dos operadores em HAskell.

Este material, junto com o que vimos nos capítulos anteriores, fornece um conjunto de ferramentas que podem ser usados para resolver problemas de programação. No próximo capítulo vamos explorar váris formas de usar essas ferramentas para resolver alguns problemas práticos.

Capítulo 4 - Projetando e

escrevendo programas

Neste capítulo nós voltaremos um passo na discussão dos detalhes do Haskell e veremos como construir programas. Apresentaremos algumas estratégias gerais para projeto de programas; que é o que falaremos sobre como programas podem ser planejados *antes* de começarmos a fazer o programa. Os conselhos que damos aqui é largamente independente do Haskell e poderá ser usado em qualquer linguagem de programação que possa ser usada.

Seguiremos discutindo recursão. Começaremos nos concentrando na explicação de **porquê** trabalhar com recursão, e seguiremos olhando **como** encontrar definições recursivas primitivas, extendendo o que temos dito sobre projeto. Concluiremos com um exame opcional de formas mais gerais de recursão.

Uma vez escrita uma definição que precisamos responder se ela faz o que ela é ao invés de o que ela faz. Concluiremos o capítulo explorando os princípios do teste de um programa e examinando um grande número de exemplos.

4.1 Por onde começamos: Projetando um programa em Haskell

Um tema que pretendo enfatizar no material é como podemos projetar programas para serem escritos em Haskell. Projeto é usado com significados muito diferentes em computação; a forma que queremos mostrar é como esta:

Definição

Projeto é o estágio antes de começar a escrever o código Haskell detalhado.

Nesta seção, nos concentraremos em olhar os exemplos, e conversar sobre as diferentes formas de definirmos funções, mas também tentaremos dar alguns conselhos sobre como começar a escrever um programa. Estas são algumas questões que podemos nós mesmo responder quando nos deparamos com um problema de programação.

Eu entendo o que preciso fazer?

Antes de começar a resolver um problema de progrmação precisamos ser claros sobre o que temos de fazer. Frequentemente os problemas são descritos em uma maneira informal, e isto pode significar que qualquer problema não esteja descrito totalmente ou não pode ser resolvido como descrito.

Suponha que queremos que a resposta seja o número do meio entre três números. É claro que dados os números 2, 4 e 3 ele deve retornar 3, mas quando apresentamos 2, 4 e 2 existe duas possibilidades de resposta.

- podemos dizer que 2 é o número do meio porque quando escrevemos os números ordenados: 2, 2 e 4, então 2 é o número que aparece no meio.
- alternativamente podemos dizer que três é o número do meio neste caso, uma vez que 2 é o menor e 4 o maior, e por isso nós não podemos retornar nenhum como resultado.

O que podemos aprender com essa ilustração?

- primeiro, essas respostas em problemas simples são coisas que temos de pensar antes de começarmos a programar
- segundo, é importante dizer que não **existe resposta correta** sobre duas opções dadas agora: a pessoa sempre espera que o programa escrito pelo programador e o que ela realmente quer sejam o mesmo.
- terceiro, uma maneira muito boa de sabermos a diferença de que o programa faz o que espera-se que faça é trabalhar com vários **exemplos**.
- Finalmente, vale a pena lembrar que dificuldades como essas surgem no estágio de programação, quando já escrevemos uma certa quantidade de definições; o quanto antes resolvermos problemas como esses, menos esforço teremos para corrigí-lo.

Outro exemplo é a definição de *max* na seção 3.4, onde dizemos que a função pode retornar quando os dois argumentos são iguais. Neste caso é sensível achar que o máximo de 3 e 3 é 3.

Posso dizer qualquer coisa sobre tipos neste estágio?

Uma coisa que podemos pensar neste estágio são os tipos. Podemos escrever

middleNumber :: Int-> Int -> Int -> Int

cin i bine e tipos da função retornando o meio de três números sem ter nenhuma idéia aobre como será feita a definição da própria função. Não obstante, é um progresso, e também é dado algo para verificarmos nossa definição novamente quando temos escrito: se tentarmos escrever uma função middleNumber mas ele não tem os tipos Int -> Int -> Int -> Int, então a função não pode ser feita.

O que já sabemos? Como posso usar esta informação?

Estas são questões cruciais para um projetista de um programa. Precisamos saber quais recursos são disponíveis para resolver o problema: quais definições já escrevemos e podem ser usadas, quais a linguagem já fornece no prelúdio e nas bibliotecas? Nós obviamente vamos aprender sobre como ir além, mas sempre quando escrever um pequeno número de programas poderemos sempre pensar sobre como isso pode nos ajudar a resolver problemas. Para instância, tentamos definir a função *maxThree* indroduzida na seção 3.4,

sabendo que já temos a função max, que fornece o máximo de dois números.

Assim, conhecendo nossos recursos também precisamos saber como podemos usá-los; isto veremos agora. Existem duas formas diferentes de uma definição já dada poder ajudar.

*Podemos ter definições de uma função como um **modelo** para o que quisermos fazer.*

Definindo maxThree temos o recurso de já ter definido a função max.

Podemos pensar na definição como um modelo para como possamos definir maxThree.

Em max temos resultado x da condição que é o máximo de dois, isto é

```
x >= y
```

Nossa definição de maxThree faz uma coisa similar, substituindo a condição para dois valores com uma condição para três:

```
x >= y && x >= z
```

Esta forma de usar max é provavelmente a primeira que nos vem à cabeça, mas não é a única e max pode nos ajudar a definir maxThree.

*Podemos **usar** uma função que já definida com uma nova definição*

Tentaremos encontrar o máximo de três números, e já contamos com uma função max que retorna o máximo de dois. Como podemos *usar* max para ter o resultado que queremos? Podemos ter o máximo dos dois primeiros e então o máximo deles com o terceiro. Na figura abaixo:

e em Haskell

```
maxThree x y z = max (max x y) z
```

ou escrevendo max na forma infixa, `max`,

```
maxThree x y z = (x `max` y) `max` z
```

Usando max dessa forma temos algumas vantagens.

A definição de maxThree é consideravelmente menor e mais legível que a original. Se em algum ponto alterarmos a forma que max é calculado - talvez criando uma função interna - então esta definição pode ter os benefícios da "nova" max. Esta não é uma grande vantagem em um exemplo pequeno como este, mas pode ser considerável em um sistema de larga escala onde podemos esperar que o software seja modificado e extendido sobre seu tempo de vida.

Eu posso quebrar o problema em pequenas partes?

Se não pudermos resolver um problema, podemos pensar em quebrá-lo em pequenas partes. Este princípio de "dividir para conquistar" é a base de toda programação em larga escala: resolvemos aspectos do problema separadamente e então juntamos tudo em uma solução global.

Como decidimos de que jeito quebraremos o problema em partes? Podemos pensar em resolver um problema simples e então criar uma solução completa, ou como podemos nós mesmos responder a essa questão.

O que fazer *se eu tenho alguma função: como usar para encontrar a solução*

Esta *O que fazer...?*é a questão central, porquê eu posso quebrar o problema em duas partes. Primeiro temos que dar uma solução *assumindo* que temos uma função auxiliar e sem saber como ela está definida. Então temos que separadamente definir estas funções auxiliares.

Ao invés de um único salto do ponto inicial para o ponto final, temos dois pequenos saltos, cada um mais fácil de fazer. Este método é chamado **top-down** como começamos do início para o topo com um problema geral, e trabalhamos quebrando em problemas menores.

Este processo pode ser feito repetidamete, de modo que o problema seja resolvido em uma série de pequenos passos. Veremos um exemplo logo a seguir; mais exemplos aparecerão no fim da seção.

Suponha que nos deparemos com o problema definido abaixo:

```
middleNumber :: Int-> Int -> Int -> Int
```

de acordo com a primeira das alterantivas decritas anteriormente nessa seção. Um modelo é dado pela definição de *maxThree*, no qual vimos a condição para x ser uma solução, y ser a solução, e assim por diante. Conseqüentemente podemos esboçar a solução:

```
middleNumber x y z  | condição para x ser a solução = x  | condição para y ser a solução = y ...
```

Agora, o problema é escrever as condições, mas aqui dizemos *o que fazer* pois nós temos a função para isso. Vamos chamá-la de *between* (entre).Ela têm três números como argumento, e um resultado Booleano,

```
between :: Int -> Int -> Int -> Bool
```

e é definida da forma **between m n p** é *True* se n está entre m e p. Agora podemos completar a definição de middleNumber:

```
middleNumber x y z  | between y x z = x  | between x y z = y  | otherwise = z
```

A definição da função between é dada como um exercício para o leitor.

Nesta seção, introduzimos algumas idéias gerais que podem ajudar no começo da resolução de um problema. Obviamente, porquê programação é uma atividade creativa não existe um conjunto de regras que sempre vai funcionar para toda resolução de problema. Por outro lado, as questões aqui ajudarão no começo, e mostrarão algumas estratégias alternativas que podem ser usadas para planejar como fazer para escrever um programa. Seguiremos essa discussão mais adiante.

Exercícios

1. Esta questão pe sobre a função
   maxFour :: Int -> Int -> Int -> Int -> Int

   que retorna o máximo de quatro inteiros. Dê 3 definições para esta função: a primeira deve ser modelada a partir de maxThree, a segunda deve usar a função max e a terceira deve usar as funções max e maxThree. Para a segunda e terceira solução, dê o diagrama que ilustra suas respostas. Discuta os méritos relativos das três soluções encontras.

2. Dada a definição da função
   between :: Int -> Int -> Int -> Bool

   discutida nesta seção. A definição deve ser consistente com o que foi dito na explicação de como middleNumber trabalha. Você também precisa ser cuidadoso com as diferentes formas que um número pode estar entre dois outros. Para facilitar, você pode usar a função

   weakAscendingOrder :: Int -> Int ->Int -> Boll

   weakAscendingOrder m n p é True exatamente quando m, n e p estão em ordem ascendente, isto é, a seqüência não decresce em nenhum ponto. Um exemplo é a seqüência 2 3 3

3. Dê uma definição para a função
   howManyEqual :: Int -> Int -> Int -> Int

   que retorna quantos dos três argumentos são iguais, assim:

   howManyEqual 34 25 36 = 0
   howManyEqual 34 25 34 = 2
   howManyEqual 34 34 34 = 3

   Lembre de quais funções você já viu - talvez nos exercícios - que podem ser usadas nesta solução.

4. Dê uma definição da função
   howManyOfFourEqual :: Int -> Int -> Int -> Int -> Int

   que é análoga de howManyEqual para quatro números. Você pode precisr pensar sobre "O que fazer...?"

4.2 Recursão

Recursão é um importante mecanismo de programação, no qual a definição e uma função ou objeto se refere ao próprio objeto. Esta seção concentra em explicar a idéia de recursão, e porque é aconselhável usá-la. Em particular daremos duas explanações complementares de como recursões primitivas trabalham na definição da função fatorial sobre números naturais. Na seção seguinte será mostrado como recursão é *usada* na prática.

***Introdução*: uma história sobre fatoriais**

Suponha que alguém nos diga que o fatorial de um número natural é o produto de todos os números naturais de 1 a este número (inclusive), ou seja, para a instância

fac 6 = 1*2*3*4*5*6

Suponha que a resposta também pode ser a tabela de fatoriais abaixo, onde temos que o fatorial de zero é um. Começaremos assim:

n	fac n
0	1
1	1
2	1*2=2
3	1*2*3=6
4	1*2*3*4=24

mas podemos notar que repetimos um conjunto de multiplicações para fazer isso. No cálculo abaixo

1*2*3*4

vemor se repetimos a multiplicação de 1*2*3 antes de multiplicar por 4.

1*2*3*4

isto sugere que podemos produzir a tabela de uma forma diferente, dizendo como começar

fac 0 = 1
Definição 1

que começa a tabela assim

n	fac n
0	1

e então dizendo como ir de uma linha até a próxima

```
fac n = fac (n-1) * n
```

Definição 2

uma vez definido, temos as linhas

n	fac n
0	1
1	1*1=1
2	1*2=2
3	2*3=6
4	6*4=24

e assim por diante.

Qual é a moral da história? Começamos descrevendo a tabela de uma forma, mas podemos ver que tudo que precisamos são as definições 1 e 2

1. (Def. 1) fornecemos a primeira linha da tabela
2. (Def. 2) mostramos como passar de uma linha da tabela para a próxima

A tabela é uma forma de escrever a função fatorial, que podemos ver nas definições 1 e 2 atualmente descrevem a **função** para cálculo do fatorial e as agrupa.

```
fac :: Int -> Int
fac n
 | n == 0 = 1
 | n > 0 = fac (n-1) * n
```

Uma definição como essa é chamada **recursiva** porque utiliza *fac* para descrever o próprio *fac*. Colocando assim, pode soar paradoxal: depois de tudo, como podemos descrever alguma coisa em termos dela mesma? Mas, depois do que já foi estudado, mostramos que a definição é perfeitamente possível, uma vez que

1. um ponto inicial: o valor de fac 0
2. uma forma de encontrar um valor de fac a partir de um ponto particular, fac (n-1), para o valor de fac na próxima linha, chamado fac n

Estas regras recursivas encontram um valor a fac n para qualquer valor positivo de n - tendo escrito n linhas da tabela, como visto.

Recursão e cálculo

Vimos na seção anterior como a definição de fatorial

```
fac :: Int -> Int
```

```
fac n
 | n == 0 = 1
 | n > 0 = fac (n-1) * n
```

pode ser vista como a geração da tabela de fatoriais, começando de fac 0 e caminhando para fac 1, fac 2 e assim por diante, até o valor desejado.

Podemos também ver a definição de uma forma calculacional, e justificar a recursão de outra forma. Tome como exemplo fac 4

```
fac 4 » fac 3 * 4
```

aplicando a definição 2, substitui um goal - fac 4 - com um goal mais simples - encontrando fac 3 (e multiplicando ele por 4). Continuando o uso da definição 2, temos

```
fac 4
» fac 3 * 4
»(fac 2 * 3) * 4
»((fac 1 * 2) * 3) * 4
»(((fac 0 * 1) * 2) * 3) * 4
```

Agora, temos o caso mais simples (ou **caso base**), que é resolvido pela definição 1.

```
»(((1 * 1) * 2) * 3) * 4
»(( 1 * 2) * 3) * 4
»( 2 * 3) * 4
» 6 * 4
24
```

No cálculo, temos trabalhado de um goal back para o caso base usando o **passo de recursão**. Podemos novamente ver que o resultado obtido é o que queremos, porque o passo de recursão vai de um caso mais complicado para um mais simples, e temos o valor do caso mais simples de todos (zero) que eventualmente chegaremos.

Veremos agora no caso do fatorial, duas explicações de porque trabalhar com recursão.

- Uma explicação *bottom-up* diz que a equação *fac* pode ser gerada para os valores de *fac* um a um até o caso base, fatorial de zero.
- uma visão *top-down* começa com um goal sendo avaliado, e mostra como simplificar as equações até o caso base.

As duas visões são relacionadas, uma vez que podemos partir da explicação top-down gerando uma tabela, mas no caso da tabela, ela é gerada como necessário. Começando com a goal do *fac 4* iremos requerer também as linhas de 0 a 3.

Tecnicamente, chamaremos a forma de recursão que teremos de **recursão primitiva**. Será descrito mais formalmente nesta seção, onde exeminamos como fazer para encontrar definições recursivas. Antes disso, será discutido outros aspectos da função *fac* definida.

Valores de erro ou indefinidos

Nossa definição de fatorial vale para zero e inteiros positivos. Qual o efeito de aplicarmos fatorial a um número negativo? Avaliando **fac (-2)** no Hugs, receberemos a mensagem de erro:

Program error: pattern match failure: fac instNum_v32 instOrd_v29 (-2)

porque fac não é definida para números negativos. Poderíamos extender a definição números negativos como zero, assim

```
fac n
 | n == 0 = 1
 | n > 0 = fac (n - 1) * n
 | otherwise = 0
```

ou podemos incluir nossa própria mensagem de erro:

```
fac n
 | n == 0 = 1
 | n > 0 = fac (n - 1) * n
 | otherwise = error "fatorial só é definido para números naturais"
```

assim, quando avaliamos fac (-2) recebemos a mensagem

Program error: fatorial só é definido para números naturais

Essa mensage é uma String, que será discutido posteriormente.

Exercícios

1. Defina a função faixaProduto que recebe os números naturais *m* e *n* e retorna o produto
   m*(m+1)*...*(n-1)*n

   Você pode incluir em usa definição o tipo de função, e sua função pode retornar 0 quando n é menor que m.
2. Como *fatorial* é um caso especial de faixaProduto, escreva uma definição de *fatorial* que usa *faixaProduto*

4.3 Recursão primitiva na prática

Esta seção examina como recursões primitivas são usadas na pratica examinando vários exemplos.

O padrão de recursão primitiva diz que você pode definir uma função de números naturais 0, 1, ... fornecendo o valor inicial,e mostrando como ir do valor n-1 ao valor n. Veremos um **modelo** pra isso

exemplo n
| n == 0 = ...
| n > 0 = ... fun (n-1) ...

onde teremos que mostrar o código dos dois lados

Como saber se a função pode ser definida dessa forma? Como visto nesse capítulo, devemos encontrar uma questão que resume a essência da necissade de se aplicar recursão primitiva.