

Welcome to GHC.IO!

```
Prelude> let fatorial a = product [1..a]
```

```
Prelude> fatorial 3
```

6

```
Prelude>
```



```
sumList [2,3,4,5]
```

```
= 2 + sumList [3,4,5]
```

```
= 2 + (3 + sumList [4,5])
```

```
= 2 + (3 + (4 + sumList [5]))
```

```
= 2 + (3 + (4 + (5 + sumList [])))
```

```
= 2 + (3 + (4 + (5 + 0)))
```

```
= 14
```

Programação Funcional

Introdução ao Haskell

Matheus Carvalho Viana

matheuscviana@ufsj.edu.br



Departamento de
Ciência da Computação



Universidade Federal
de São João del-Rei

História

A primeira linguagem funcional foi o LISP, criada no final da década de 1950.

De lá até a década de 1980 surgiram diversas linguagens funcionais, mas muitas delas não eram “puras”.

Em 1987, foi realizada a conferência *Functional Programming Languages and Computer Architecture* (FPCA '87), visando definir um padrão comum para as linguagens funcionais.

A partir disso, surgiu um comitê para a criação de uma linguagem funcional que, deveria seguir algumas especificações:

- Ser viável para o ensino, pesquisa e aplicações, incluindo sistema de larga escala;
- Possuir uma sintaxe e semântica descritiva;
- Não ser proprietária, para que qualquer um pudesse implementá-la e distribuí-la;
- Basear-se em ideias que envolvessem o senso comum;
- Deveria reduzir a diversidade desnecessária de outras linguagens funcionais (ser puramente funcional).

Introdução

A partir desse comitê, surgiu Haskell, uma linguagem de programação puramente funcional, de propósito geral, fortemente tipada, nomeada em homenagem ao matemático Haskell Curry.

A primeira versão foi definida em 1 de abril de 1990.

Outras linguagens como R, Octave, MatLab, Maple e Mathematica também se aproximam muito da matemática, mas essas últimas são de propósito específico.

Ao invés de termos algoritmos expressos através em uma sequência de instruções, como ocorre nas linguagens imperativas, Haskell tem seus algoritmos totalmente baseados no conceito matemático de função.



Características

Possui suporte a:

- funções recursivas;
- tipagem estática e forte;
- casamento de padrões;
- compreensão de lista;
- sentenças de guarda;
- avaliação preguiçosa, entre outros recursos.

A combinação destas características pode fazer com que a construção de funções que seriam complexas em uma linguagem imperativa torne-se uma tarefa quase trivial em Haskell.

Estrutura do Código

A extensão dos arquivos Haskell é **.hs**.

O nome do arquivo pode ser qualquer um, desde que não tenha espaço nem caracteres especiais.

Existe tanto um compilador (GHC) quanto um interpretador (GHCi) para executar o código. Para compilar, é necessário que haja uma função **main**.

Essa função sempre retorna uma saída do tipo `IO()` e a forma mais simples de obter isso é usando a função **print**.

```
main = print (div 30 10)
```

Estrutura do Código

Haskell usa um esquema chamado *layout* para estruturar seu código, a exemplo de Python.

Isso permite que o código seja escrito sem delimitadores sintáticos, como ponto e vírgula e chaves, mas também obriga que a indentação do texto siga regras fundamentais:

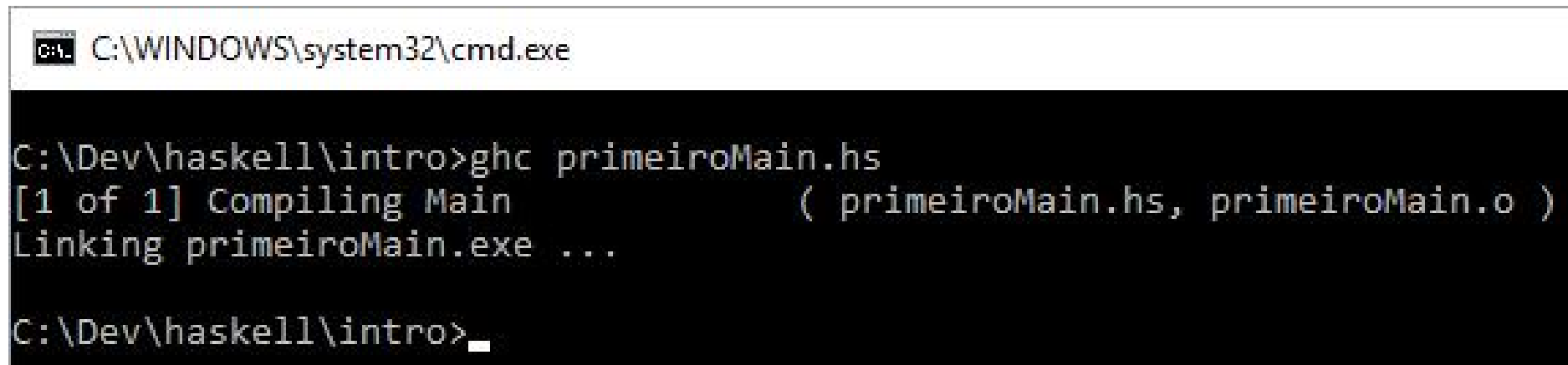
- se uma linha começa em pelo menos uma coluna à frente (mais à direita) do início da linha anterior, é considerada a continuação da linha precedente;
- se uma linha começa na mesma coluna que a do início da linha anterior, elas são consideradas definições independentes entre si;
- se uma linha começa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

Nunca use tab nos arquivos Haskell.

Compilador do Haskell

A principal ferramenta de suporte ao desenvolvimento com Haskell é o Glasgow Haskell Compiler (GHC).

Ele pode ser acionado via terminal a partir do comando `ghc` junto com o nome do arquivo. Essa opção só é válida se o código-fonte possuir uma função `main`.



```
C:\WINDOWS\system32\cmd.exe

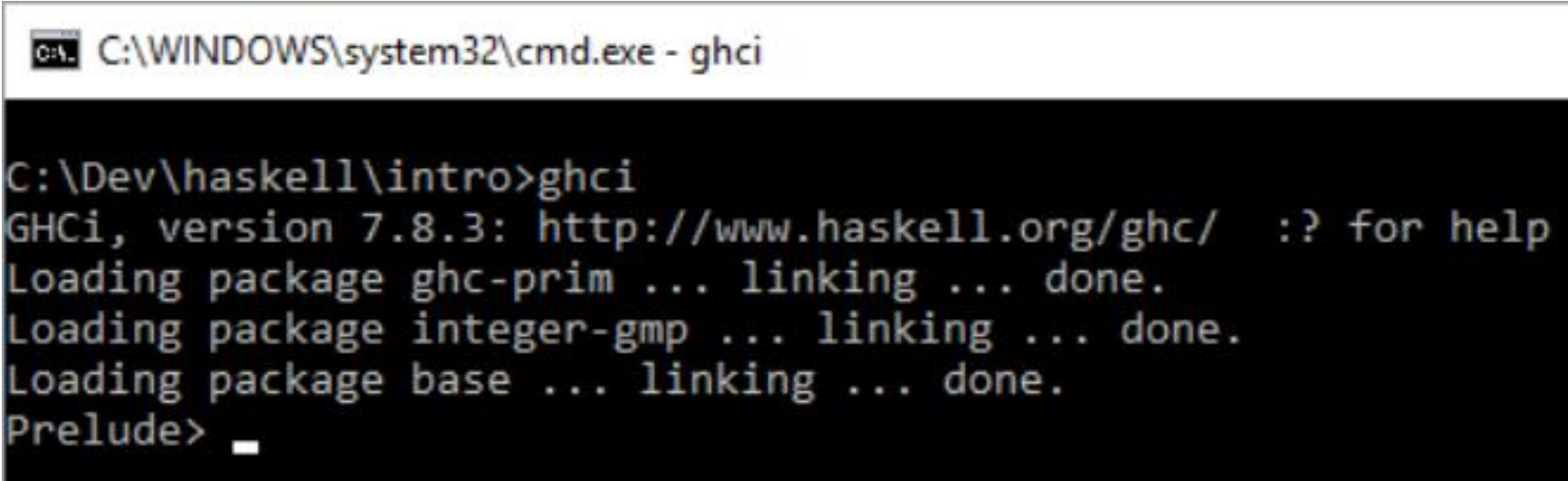
C:\Dev\haskell\intro>ghc primeiroMain.hs
[1 of 1] Compiling Main                ( primeiroMain.hs, primeiroMain.o )
Linking primeiroMain.exe ...

C:\Dev\haskell\intro>_
```

Interpretador do Haskell

Existe uma outra opção para criar programas Haskell, usando o interpretador **GHCI**.

Nesse caso, não é obrigatória a existência da função main.



```
C:\WINDOWS\system32\cmd.exe - ghci

C:\Dev\haskell\intro>ghci
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> _
```


Interpretador do Haskell

Principais comandos do GHCi:

:help ou **:?** – comando de ajuda que mostra a lista de comandos.

:quit ou **:q** ou **Ctrl+d** – fecha o GHCi;

:load ou **:l** e o nome de um arquivo .hs – carrega o arquivo indicado para usar as funções definidas nele;

:reload ou **:r** – recarrega o arquivo atualmente carregado;

:edit ou **:e** – abre o arquivo atual em um editor de texto. Se usado junto com um nome de um arquivo, abre esse arquivo indicado;

:main – executa a função main, se houver;

:module ou **:m** e o nome de uma biblioteca ou módulo – carrega a biblioteca ou módulo indicado;

: – executa novamente o último comando executado.

Interpretador Haskell

É possível usar várias expressões matemáticas no GHCi e ter uma resposta.

Prelude> é o prompt padrão do GHCi.

```
Prelude> 3 * 5
```

```
15
```

```
Prelude> 4 ^ 2 - 1
```

```
15
```

```
Prelude> (1 - 5)^(3 * 2 - 4)
```

```
16
```

Interpretador Haskell

Strings são escritas em "aspas".

A concatenação de strings é feita pelo operador ++.

```
Prelude> "Oi"
```

```
"Oi"
```

```
Prelude> "Oi" ++ ", Haskell"
```

```
"Oi, Haskell"
```

Interpretador Haskell

Existem diversas funções pré-carregadas no Haskell.

Normalmente não há parênteses nas chamadas, mas é necessário quando um parâmetro é o resultado de uma função ou operação.

```
Prelude> succ 5
```

```
6
```

```
Prelude> truncate 6.59
```

```
6
```

```
Prelude> round 6.59
```

```
7
```

```
Prelude> sqrt 2
```

```
1.4142135623730951
```

```
Prelude> not (5 < 3)
```

```
True
```

```
Prelude> gcd 21 14
```

```
7
```

Interpretador Haskell

Ações de I/O (entrada e saída) podem ser usadas para ler e escrever no console. Algumas delas que são comuns incluem:

```
Prelude> putStrLn "Oi, Haskell!"
```

```
Oi, Haskell!
```

```
Prelude> putStrLn "Sem quebra de linha!"
```

```
Sem quebra de linha!
```

```
Prelude> print (5 + 4)
```

```
9
```

```
Prelude> print (1 < 2)
```

```
True
```

Interpretador Haskell

Leituras podem ser feitas com a função `readLn`.

Um bloco `do` define uma região de I/O.

O símbolo `<-` é usado para atribuir um nome ao resultado de uma ação de I/O.

```
Prelude> do { n <- readLn ; print (n^2) }
```

4

16

4 é o número digitado e 16 é a saída do print.

Interpretador Haskell

É possível definir uma função no Prelude do GHCi, para usá-la posteriormente.
Para isso, use o comando **let**.

```
Prelude> let areaq lado = lado^2
```

```
Prelude> areaq 4
```

```
16
```

Importação de Biblioteca

A biblioteca padrão do Haskell é chamada de **Prelude**.

Ela é carregada automaticamente, então não é necessário importá-la para usar as suas funções. Por isso, o nome dela vem escrito quando o GHCi é ligado.

```
GHCi, version 7.8.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```


Importação de Biblioteca

O Haskell é uma linguagem com um biblioteca muito rica. Muitas funções úteis já vem carregadas com o Prelude.

Porém, há muitas outras funções úteis também que não pertencem ao Prelude e, para utilizá-las, é preciso importar a biblioteca delas.

A tabela abaixo apresenta alguns exemplos de bibliotecas:

Biblioteca	Descrição
Prelude	Fornece funções comuns para manipulação de dados.
Data.Char	Fornece funções para manipulação e conversão de caracteres.
Data.String	Fornece funções para manipulação e conversão de Strings.
Data.List	Provê funções que trabalham sobre listas.
System.IO	Provê um conjunto de tipos e funções básicos para tratamento de IO.

Importação de Biblioteca

Para importar uma biblioteca, usa-se o comando **import**, que pode ser usado tanto dentro de um arquivo quanto na interface do GHCi.

Exemplo de como fazer no arquivo:

```
import Data.Char
```

Exemplo de como fazer no GHCi:

```
Prelude> import Data.Char
```

```
Prelude Data.Char>
```

Importação de Biblioteca

No GHCi, também é possível usar o comando **:module** ou **:m** para carregar uma biblioteca. A diferença é que **:module** mantém somente a última biblioteca carregada, além de Prelude. Já **import** mantém todas as bibliotecas carregadas até o momento.

```
Prelude> import Data.Char  
Prelude Data.Char> import Data.String  
Prelude Data.Char Data.String>
```

O comando **import** manteve todas as bibliotecas carregadas.

```
Prelude Data.Char Data.String> :module Data.List  
Prelude Data.List>
```

O comando **:module** manteve somente Prelude e a mais nova.

Importação de Biblioteca

Em alguns casos, ao invés de importar todo o conteúdo de uma biblioteca, é interessante importar somente as funções ou tipos dela que vamos realmente usar.

Isso é extremamente útil para evitar conflitos nos casos em que o nosso código define uma função ou tipo que tem o mesmo nome de uma função ou tipo da biblioteca carregada.

Também é bom para diminuir o tamanho do arquivo compilado.

Importação de Biblioteca

Para importar somente uma função ou tipo de uma biblioteca, basta indicar o nome dela entre parênteses após o comando import.

Isso pode ser feito, da mesma maneira, tanto no GHCi quanto dentro de um arquivo.

```
Prelude> import Data.Char (ord)
```

Somente a função
ord foi importada.

```
Prelude Data.Char> ord 'a'
```

Assim foi possível usá-la.

```
97
```

```
Prelude Data.Char> chr 65
```

A função chr não foi importada,
assim o GHCi não a reconheceu.

```
<interactive>:10:1: Not in scope: 'chr'
```

```
Prelude Data.Char> import Data.Char (ord,chr)
```

```
Prelude Data.Char> chr 65
```

```
'A'
```

```
Prelude Data.Char>
```

Assim, ord e chr foram
importadas.

Importação de Biblioteca

Também é possível importar toda a biblioteca, exceto algumas funções ou tipos. Para isso usa-se o comando **hiding** junto ao import..

```
Prelude> import Data.Char hiding (chr)
Prelude Data.Char> ord 'a'
97
Prelude Data.Char> chr 97

<interactive>:12:1: Not in scope: 'chr'
Prelude Data.Char>
```

Todo o conteúdo da biblioteca Data.Char foi importada, exceto a função chr.

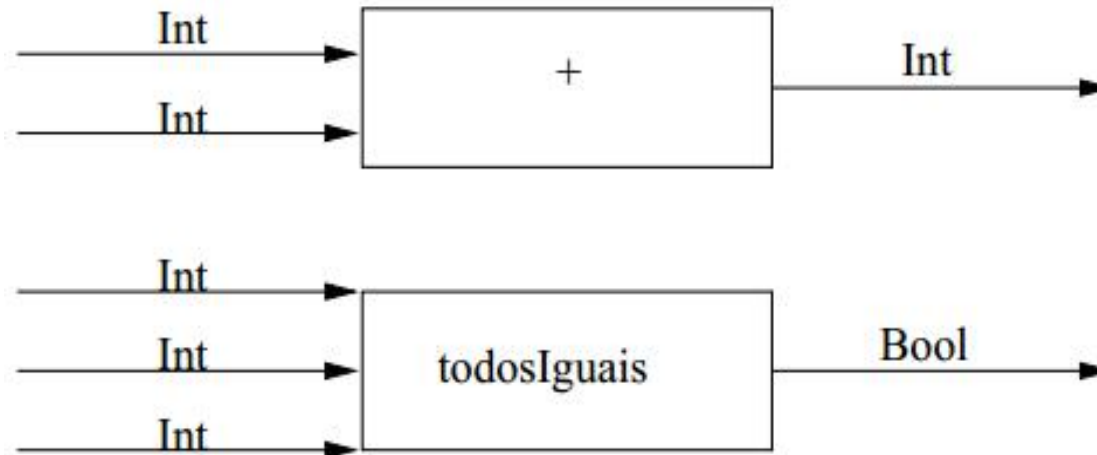
A função chr não foi importada, assim o GHCi não a reconheceu.

Funções

A estrutura principal dos programas Haskell são as funções.

A definição de função em Haskell provém da matemática, em que existe um conjunto domínio de entrada e um conjunto imagem como saída.

As funções podem ser representadas graficamente por uma caixa que recebe um ou mais parâmetros como entrada (argumentos), processa-os e constrói um resultado único que é exibido como saída.



Funções

A sintaxe de uma função é:

```
nomeDaFunção :: TipoDoParâmetro -> ... -> TipoDeRetorno  
nomeDaFuncao nomeDoParâmetro ... = codigoDaFunção
```

Exemplo:

```
soma3Int :: Int -> Int -> Int -> Int  
soma3Int a b c = a + b + c
```

A primeira linha das funções é responsável para definir a quantidade de parâmetros de entrada, o tipo de cada um e também o tipo da saída (último tipo informado).

Essa primeira linha pode ser omitida. Nesse caso, Haskell realiza uma **inferência de tipos**, ou seja, determina automaticamente o tipo de entrada com base nas operações que são realizadas na função.

Funções

O símbolo = não é uma atribuição. Ele significa "é definida por" ou "tem como equação".

O entendimento é o mesmo das funções da matemática.

Matemática	Haskell
$f(x) = x + x$	$f\ x = x + x$

Em ambos os casos, o significado é o mesmo: a função f para x tem como equação $x + x$.

Pode-se entender também como: a função f , dada uma entrada x , tem como resultado $x + x$.

Funções

Exemplos de funções em Haskell:

`x = 5`

Função x, sem entrada, com resultado sempre 5.

`area l a = l * a`

Função area, com entradas l e a, cujo resultado é equação $l * a$.

`fdex x = x^2 + 3*x + 1`

Função fdex, com entrada x, cujo resultado é equação
 $x^2 + 3x + 1$.

Forma de Avaliação do Código

As funções são avaliadas na ordem leftmost-outermost, usando um mecanismo de avaliação preguiçosa, que só avalia uma expressão se ela for realmente necessária e no máximo uma vez.

Tomando as seguintes funções como exemplo:

```
todosIguais a b c = (a == b) && (b == c)
quadrado x = x * x
```

A expressão:

```
todosIguais (quadrado 3) (sqrt 4) (quadrado 2)
```

É avaliada na seguinte ordem:

```
((quadrado 3) == (sqrt 4)) && ((sqrt 4) == (quadrado 2))
((3 * 3) == (sqrt 4)) && ((sqrt 4) == (quadrado 2))
(9 == (sqrt 4)) && ((sqrt 4) == (quadrado 2))
(9 == 2) && (2 == (quadrado 2))
False && (2 == (quadrado 2))
False
```

Casamento de Padrões

As entradas das funções podem ser definidas com entradas constantes e chama a versão correspondente da função quando há um casamento entre entrada constante e o valor passado na chamada.

```
fatorial :: Int -> Int
```

```
fatorial 0 = 1
```

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

Console:

```
Prelude> fatorial 5  
120
```

fatorial 5 casa com essa
versão da função.

```
Prelude> fatorial 0  
1
```

fatorial 0 casa com essa
versão da função.

Casamento de Padrões

O símbolo *underline* () pode ser usado para indicar as situações em que uma entrada não interfere no resultado.

```
eLogico :: Bool -> Bool -> Bool
```

```
eLogico True True = True
```

```
eLogico False _ = False
```

```
eLogico _ False = False
```

```
funcao :: Double -> Double -> Double
```

```
funcao 0 _ = 1
```

```
funcao x y = sin x / x * y
```

```
fib :: Int -> Int
```

```
fib 1 = 0
```

```
fib 2 = 1
```

```
fib n = fib (n-2) + fib (n-1)
```

Nomes

Haskell é **sensível ao caso**, ou seja, diferencia letras maiúsculas de minúsculas.

Nomes de tipos de dados sempre devem ser iniciados por uma letra maiúscula. Os demais nomes devem ser iniciados com letra minúscula. Isso é regra, não convenção como ocorre em C e Java.

A primeira letra dos nomes pode ser seguida de outros caracteres, que podem ser letras maiúsculas ou minúsculas (acentuadas ou não), c cedilha, dígitos, sublinhado ou apóstrofo (').

```
função' :: Int -> Int
```

```
função' x = -x
```

Nomes

As palavras reservadas da linguagem são sempre escritas em letras minúsculas.

Haskell apresenta 22 palavras reservadas:

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

Operadores

Operadores são semelhantes às funções, porém são infixos por padrão e são invocados por meio de símbolo especial ao invés de um nome.

Em geral, os operadores executam operações simples da matemática ou lógica.

Cada tipo de dado possui um conjunto de operadores válidos para os seus valores.

Tipos de Dados

Tipo de dado é uma propriedade dos valores que podem ser trabalhados pelas linguagens de programação.

Serve para definir:

- Um conjunto de valores válidos;
- Um conjunto de operações válidas para um determinado valor;
- A quantidade de espaço ocupada em memória por um valor/variável;
- A forma de interpretação de um valor armazenado na memória.

Tipos de Dados

Haskell é uma linguagem com

- **tipagem estática explícita**, pois é necessário informar qual o tipo das variáveis e não é possível alterar esse tipo durante a execução do programa, e
- **fortemente tipada**, pois não é possível fazer coerção implícita (converter automaticamente de um tipo para outro, como em C).

```
soma :: Int -> Int -> Int  
soma a b = a + b
```

A entrada 5.5 não é do tipo inteiro.
Assim o programa acusa um erro.

```
*Main> soma 5.5 2
```

```
<interactive>:11:6:
```

```
No instance for (Fractional Int) arising from the literal '5.5'
```

```
In the first argument of 'soma', namely '5.5'
```

```
In the expression: soma 5.5 2
```

```
In an equation for 'it': it = soma 5.5 2
```

Principais Tipos de Dados

Tipo	Descrição	Exemplo
Bool	Enumeração de valores booleanos, que permitem certas operações lógicas, como conjunção (&&), disjunção () e negação (not).	True False
Char	Enumeração de caracteres 16-bit, Unicode. A faixa dos primeiro 128 caracteres é idêntica ao ASCII.	'a'
Double	Ponto flutuante com maior intervalo e precisão que Float	321.6e-3
Float	Ponto flutuante	6553.61
Int	Inteiro que cobre, pelo menos, o intervalo de valores $[-2^{29}, 2^{29} - 1]$.	123
Integer	Inteiro de precisão ilimitada, com as mesmas funções e operadores de Int	123
Rational	Número fracionário de precisão ilimitada formado pela razão entre dois inteiros usando o operador % (biblioteca Data.Ratio)	1 % 3
Word	Inteiro sem sinal de precisão fixa.	123
()	Pronuncia-se "unit". Equivale ao tipo void do C.	()

Principais Tipos de Dados

Principais funções e operadores da biblioteca padrão Prelude para trabalhar com os tipos numéricos (Int, Float, etc.):

+ - *	Operadores binários de adição, subtração e multiplicação.	<code>6 + 5 == 11</code>
-	Operador unário para definir números negativos, que devem ficar entre parênteses quando usados em expressões.	<code>-10</code> <code>3 * (-3) == (-9)</code>
abs	Valor absoluto do número (sem sinal).	<code>abs (-5) == 5</code>
even	Retorna True se o número for par e False se for ímpar.	<code>even 2 == True</code>
max	Retorna o maior entre dois números.	<code>max 5 2 == 5</code>
min	Retorna o menor entre dois números.	<code>min 5 2 == 2</code>
negate	Inverte o sinal .	<code>negate 5 == (-5)</code>
odd	Retorna True se o número for ímpar e False se for par.	<code>odd 3 == True</code>
pred	Valor predecessor. Para tipos numéricos, subtrai 1.	<code>pred 4 == 3</code>
show	Converte de número para String.	<code>show 5 == "5"</code>
signum	-1 para número negativo, 0 para zero e 1 para positivo.	<code>signum 4 == 1</code>
succ	Valor sucessor. Para tipos numéricos, adiciona 1.	<code>succ 4 == 5</code>

Principais Tipos de Dados

Principais funções e operadores da biblioteca padrão Prelude para trabalhar com Int, Word e Integer:

^	Potência de inteiros. Resulta em um tipo inteiro.	<code>4 ^ 2 == 16</code>
div	Divisão de inteiros tendendo ao infinito negativo.	<code>div 3 2 == 1</code> <code>div (-3) 2 == (-2)</code>
fromEnum	Converte um valor numérico ou Char para Int.	<code>fromEnum 'a' == 65</code>
fromIntegral	Converte qualquer tipo inteiro no tipo da expressão em que foi usada.	<code>5.5 + fromIntegral 5 == 10.5</code>
mod	Resto da divisão de inteiros tendendo ao infinito negativo.	<code>mod 3 2 == 1</code> <code>mod (-3) 2 = 1</code>
quot	Divisão de inteiros tendendo a zero.	<code>quot 3 2 == 1</code> <code>quot (-3) 2 == (-1)</code>
rem	Resto da divisão de inteiros tendendo a zero.	<code>rem 3 2 == 1</code> <code>rem (-3) 2 = -1</code>
toEnum	Converte um Int (apenas) no tipo da expressão em que foi usada.	<code>5.5 + toEnum 5 == 10.5</code>

Principais Tipos de Dados

Principais funções e operadores da biblioteca Prelude para trabalhar com Float e Double:

^^	Potência de inteiro com ponto flutuante.	<code>5 ^^ 2 == 25.0</code>
**	Potência de ponto flutuante.	<code>5 ** 2 == 25.0</code>
acos, asin, atan	Cosseno, seno e tangente inversos.	<code>acos 0 == 1.5707</code>
ceiling	Retorna o menor inteiro não menor que o número passado.	<code>ceiling 1.9 == 2</code> <code>ceiling (-1.9) == (-1)</code>
cos, sin, tan	Cosseno, seno e tangente.	<code>cos 0 == 1.0</code>
exp, log	Potência e logaritmo neperianos.	<code>exp 1 == 2.7182</code>
floor	Retorna o maior inteiro não maior que o número passado.	<code>floor 1.9 == 1</code> <code>floor (-1.9) == (-2)</code>
logBase	Logaritmo.	<code>logBase 4 16 == 2.0</code>
pi	Função que retorna o valor da constante π .	<code>pi == 3.1415</code>
round	Retorna o inteiro mais próximo do número passado, ou o inteiro par se equidistantes.	<code>round 1.1 == 1</code> <code>round 1.5 == 2</code>
sqrt	Raiz quadrada.	<code>sqrt 4 == 2.0</code>
truncate	Retorna o inteiro mais próximo entre 0 e número passado.	<code>truncate 1.9 == 1</code> <code>truncate (-1.9) == (-1)</code>

Principais Tipos de Dados

Principais funções da biblioteca Prelude:

pred	Obtém o caractere predecessor.	<code>pred 'd' == 'c'</code>
show	Converte de Char para String.	<code>show '5' == "'5'"</code>
succ	Obtém o caractere sucessor.	<code>succ 'c' == 'd'</code>

Principais funções da biblioteca Data.Char:

chr	Converte um código ASCII (Int) para o seu Char correspondente.	<code>chr 65 == 'a'</code>
digitToInt	Converte um dígito (Char) para Int.	<code>digitToInt '1' == 1</code>
ord	Converte um Char para seu código ASCII (Int).	<code>ord 'a' == 65</code>

Principais Tipos de Dados

Assim como no C, o tipo String é na verdade uma lista do tipo Char.

Principais funções da biblioteca Prelude para trabalhar com o tipo String:

++	Concatena duas Strings.	<code>"ab" ++ "cd" == "abcd"</code>
:	Adiciona um Char ao início de uma String.	<code>'a' : "bc" == "abc"</code>
read	Converte uma String no tipo da expressão em que foi usada.	<code>5 + read "7" == 12</code>
show	Converte qualquer tipo primitivo para String.	<code>show 1 == "1"</code>

Operador de Indicação de Tipo

Haskell aplica inferência de tipo nas expressões, ou seja, ele adivinha o tipo de uma expressão com base no contexto em que ela ocorre.

Por exemplo, na expressão `5 + read "7"`, a função `read` percebe que deve converter a `String "7"` para o inteiro `7` porque ela é usada no contexto de uma soma de inteiros.

```
Prelude> 5 + read "7"
```

```
12
```

Operador de Indicação de Tipo

Mas há situações em que não é possível inferir o tipo de uma expressão porque o contexto não está claro.

Por exemplo, se usarmos a função `read` sozinha no Prelude, o interpretador acusa um erro, pois ele não consegue determinar para qual tipo deve converter a String `"7"`.

```
Prelude> read "7"
```

```
*** Exception: Prelude.read: no parse
```

Para situações como essa é necessário usar o operador `::`.

```
Prelude> read "7" :: Int
```

```
7
```

`::` é um operador binário, cujo primeiro operando é uma expressão e o segundo é o tipo para o qual deve converter o valor da expressão.

Operadores Relacionais

No Haskell, podem ser aplicados para todos os tipos primitivos e também para as listas e String.

>	Maior que
>=	Maior ou igual
==	Igual
/=	Diferente
<=	Menor ou igual
<	Menor

Cuidado que não é !=
como em C e Java.

Operadores Booleanos

Os únicos valores booleanos são **True** e **False** e sobre eles podem ser utilizadas funções pré-definidas ou funções construídas pelo usuário.

Função	Nome	Tipo
&&	and	$\&\& :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$
	or	$:: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$
not	inversor	$\text{not} :: \text{Bool} \rightarrow \text{Bool}$

Operadores vs Funções

Qualquer operador pode ser usado como função (prefixa), e qualquer função pode ser usada como um operador (infixo).

Basta incluir o operador entre parênteses () e a função entre crases ``.

<code>Prelude> 2 + 3</code>	--usando operador na forma infixa
<code>Prelude> (+) 2 3</code>	--usando operador como função (prefixo)
<code>Prelude> div 4 2</code>	--usando função na forma prefixa
<code>Prelude> 4 `div` 2</code>	--usando função como operador (infixa)

Criação de Operadores

Haskell permite aos programadores inventarem operadores binários utilizando 2 ou mais dos seguintes símbolos:

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

```
(!=) :: Int -> Int -> Bool
```

```
a != b = a /= b
```

```
(//) :: Int -> Int -> Int
```

```
a // b = div a b
```

Criação de Operadores

Ao definir um operador, o ideal é indicar sua precedência e associatividade.

Nível	Operador
9	. !!
8	^ ^^ **
7	* / `div` `quot` `rem` `mod`
6	+ -
5	: ++
4	== /= < <= > >= `elem` `notElem`
3	&&
2	
1	Não utilizado no Prelude

Associatividade	
Esquerda	infixl
Direita	infixr
Nenhuma	infix

Exemplos:

```
infix 4 !=  
infixl 7 //
```

Exercícios

1. Implemente a função **nAnd** `:: Bool -> Bool -> Bool` que dá o resultado **True**, exceto quando seus dois argumentos são **True**.
2. Implemente a função **fibonacci** que recebe uma posição (inteiro) e retorna o valor (inteiro) na posição indicada da série.
3. Implemente uma função que calcula a soma de inteiros não negativos usando as funções **sucessor** (**succ**) e **predecessor** (**pred**).
4. Transforme a função da questão 1 em um operador.
5. Crie um operador **==** compara um inteiro de um algarismo com um dígito. Ex.: `1 == '1'` retorna **True**.
6. Crie uma função que calcula a área de um círculo a partir do seu raio.
7. Qual é a diferença entre `x = 3` e `x == 3` no Haskell?
8. O que está errado com a definição da função **todosDiferentes** abaixo?
todosDiferentes `n m p = (n /= m) && (m /= p)`