# Folha de Referência de Haskell Básico

## Estrutura

```
func :: type -> type
func x = expr

fung :: type -> [type] -> type
fung x xs = expr

main = do code
          code
          ...
```

## Aplicação de Funções

```
f x y       ≡ (f x) y          ≡ ((f) (x)) (y)
f x y z     ≡ ((f x) y) z      ≡ (f x y) z
f $ g x     ≡ f (g x)          ≡ f . g $ x
f $ g $ h x ≡ f (g (h x))      ≡ f . g . h $ x
f $ g x y   ≡ f (g x y)        ≡ f . g x $ y
f g $ h x   ≡ f g (h x)        ≡ f g . h $ x
```

## Valores e Tipos

| é do tipo | *expr* | :: *tipo* |
|---|---|---|
| booleano | `True \|\| False` | `:: Bool` |
| caracter | `'a'` | `:: Char` |
| inteiro de precisão fixa | `1` | `:: Int` |
| inteiro (tam. arbitrário) | `31337` | `:: Integer` |
| | `31337^10` | `:: Integer` |
| real de precisão simples | `1.2` | `:: Float` |
| real de precisão dupla | `1.2` | `:: Double` |
| lista | `[]` | `:: [a]` |
| | `[1,2,3]` | `:: [Integer]` |
| | `['a','b','c']` | `:: [Char]` |
| | `"abc"` | `:: [Char]` |
| | `[[1,2],[3,4]]` | `:: [[Integer]]` |
| string | `"asdf"` | `:: String` |
| tupla | `(1,2)` | `:: (Int,Int)` |
| | `([1,2],'a')` | `:: ([Int],Char)` |
| relacional de ordem | `LT, EQ, GT` | `:: Ordering` |
| função ($\lambda$) | `\x -> e` | `:: a -> b` |
| pode ser (apenas algo ou nada) | `Just 10` | `:: Maybe Int` |
| | `Nothing` | `:: Maybe a` |

## Valores e Classes de tipos (Typeclasses)

| dado contexto, é do tipo | *expr* | :: *restrição* => *tipo* |
|---|---|---|
| Numérico (+,-,*) | `137` | `:: Num a => a` |
| Fracional (/) | `1.2` | `:: Fractional a => a` |
| Real | `1.2` | `:: Floating a => a` |
| Comparável (==) | `'a'` | `:: Eq a => a` |
| Ordenável (<=,>=,>,<) | `731` | `:: Ord a => a` |

## Declaração de Tipo e Classes

sinônimo de tipo
```
type MyType = Type
type PairList a b = [(a,b)]
type String = [Char]      -- from Prelude
```

dado (construtor único)
```
data MyData = MyData Type Type
  deriving (Class, Class)
```

dado (múltiplos construtores)
```
data MyData = Simple Type
            | Duple Type Type
            | Nople
```

dado (sintaxe de registro)
```
data MDt = MDt { fieldA
               , fieldB :: TyAB
               , fieldC :: TyC }
```

novo tipo (newtype)
↳ (único constr./campo)
```
newtype MyType = MyType Type
  deriving (Class, Class)
```

typeclass
```
class MyClass a where
  foo :: a -> a -> b
  goo :: a -> a
```

instância de typeclass
```
instance MyClass MyType where
  foo x y = ...
  goo x = ...
```

## Operadores (agrupados por precedência)

| Índice de lista, função composição | `!!`, | `.` |
|---|---|---|
| elevado a: Não-neg. Int, Int, Float | | `^`, `^^`, `**` |
| multiplicação, divisão fracional | `*`, `/` | |
| divisão integral ($\Rightarrow -\infty$), módulo | `` `div` ``, `` `mod` `` | |
| quociente integral ($\Rightarrow 0$), resto | `` `quot` ``, `` `rem` `` | |
| adição, subtração | `+`, `-` | |
| construção de lista, append em lista | | `:`, `++` |
| diferença entre lista | | `\\` |
| comparações: | `>`, `>=`, `<`, `<=`, `==`, `/=` | |
| pertinência a lista (membership) | `` `elem` ``, `` `notElem` `` | |
| booleano and | | `&&` |
| booleano or | | `\|\|` |
| sequenciação: bind e then | `>>=`, `>>` | |
| aplicação, aplic. estrita, sequenciação | `$`, `$!`, `` `seq` `` | |

NOTA: Mais alta precedência (primeira linha) é 9, mais baixa é 0. Operadores listados alinhados à esquerda, direita, e centro indicam associatividade à esquerda, direta, e não existente.

**Def. fixidade:**

| | |
|---|---|
| não associativa | `infix 0-9 'op'` |
| associativa esquerda | `infixl 0-9 +-+` |
| associativa à direita | `infixr 0-9 -!-` |
| default (quando não inform.) | `infixl 9` |

## Funções ≡ Operadores Infixos

```
f a b   ≡ a `f` b
a + b   ≡ (+) a b
(a +) b ≡ ((+) a) b
(+ b) a ≡ (\x -> x + b) a
```

## Expressões / Cláusulas

**expressão if** ≈ **equações com guardas**

```
if boolExpr                foo ... | boolExpr  = exprA
  then exprA                       | otherwise = exprB
  else exprB
```

**expressão if aninhada** ≈ **equações com guardas**

```
if boolExpr1               foo ... | boolExpr1 = exprA
  then exprA                       | boolExpr2 = exprB
  else if boolExpr2                | otherwise = exprC
         then exprB
         else exprC
```

**expressões case** ≈ **função casamento de padrões**

```
case x of pat1 -> exA      foo pat1 = exA
          pat2 -> exB      foo pat2 = exB
          _    -> exC      foo _    = exC
```

**expressão case com 2 variáveis** ≈ **função casamento de padrões**

```
case (x,y) of              foo pat1 patA = exprA
  (pat1,patA) -> exprA     foo pat2 patB = exprB
  (pat2,patB) -> exprB     foo _    _    = exprC
  _           -> exprC
```

**expressões let** ≈ **cláusula where**

```
let nameA = exprA          foo ... = mainExpression
    nameB = exprB            where nameA = exprA
in mainExpression                  nameB = exprB
```

**notação do** ≈ **notação do sem açucar**

```
do patA <- action1         action1 >>= \patA ->
   action2                   action2 >>
   patB <- action3           action3 >>= \patB ->
   action4                     action4
```

## Casam. de Padrões (fn. declaração, lambda, case, let, where)

| | | | | |
|---|---|---|---|---|
| **fixa** | número 3 | 3 | character 'a' | 'a' |
| | ignorar valor | _ | empty string | |
| **lista** | vazia | | | [] |
| | "head"x e "tail"xs | | | (x:xs) |
| | "tail"xs (ignorar "head") | | | (_:xs) |
| | lista com 3 elementos | | | [a,b,c] |
| | lista onde 2o. elemento é 3 | | | (x:3:xs) |
| **tupla** | par valores a e b | | | (a,b) |
| | ignorar segundo elemento | | | (a,_) |
| | valores triplos a, b and c | | | (a,b,c) |
| **mista** | prim. tupla na lista | | | ((a,b):xs) |
| **maybe** | apenas construtor | | | Just a |
| | contrutor nothing | | | Nothing |
| **personal.** | tipo def. pelo usuário | | | MyData a b c |
| | ignorar segundo campo | | | MyData a _ c |
| | tipo registro def. pelo usuário | | | MyR { f1=x, f2=y } |
| **como padrão** | tupla s e seus valores | | | s@(a,b) |
| | lista a, sua "head"e "tail" | | | a@(x:xs) |

## Miscelânia

```
id         :: a -> a                id x  ≡  x   -- identidade
const      :: a -> b -> a    (const x) y  ≡  x
undefined  :: a                undefined  ≡  ⊥ (lança erro)
error      :: [Char] -> a       error cs  ≡  ⊥ (lança erro cs)
not        :: Bool -> Bool      not True  ≡  False
flip       :: (a -> b -> c) -> b -> a -> c
                        flip f $ x y  ≡  f y x
```

## Listas

```
null :: [a] -> Bool                  null []  ≡  True  -- ∅?
length :: [a] -> Int           length [x,y,z]  ≡  3
elem :: a -> [a] -> Bool      y `elem` [x,y]  ≡  True  -- ∈?
head :: [a] -> a               head [x,y,z,w]  ≡  x
last :: [a] -> a               last [x,y,z,w]  ≡  w
tail :: [a] -> [a]             tail [x,y,z,w]  ≡  [y,z,w]
init :: [a] -> [a]             init [x,y,z,w]  ≡  [x,y,z]
reverse :: [a] -> [a]         reverse [x,y,z]  ≡  [z,y,x]
take :: Int -> [a] -> [a]      take 2 [x,y,z]  ≡  [x,y]
drop :: Int -> [a] -> [a]      drop 2 [x,y,z]  ≡  [z]
takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]
                takeWhile (/= z) [x,y,z,w]  ≡  [x,y]
zip  :: [a] -> [b] -> [(a, b)]
                   zip [x,y,z] [a,b]  ≡  [(x,a),(y,b)]
```

### Listas Infinitas

```
repeat  :: a -> [a]      repeat x  ≡  [x,x,x,x,x,x,...]
cycle   :: [a] -> [a]    cycle xs  ≡  xs++xs++xs++...
                       cycle [x,y]  ≡  [x,y,x,y,x,y,...]
iterate :: (a -> a) -> a -> [a]
                    iterate f x  ≡  [x,f x,f (f x),...]
```

### Alta-ordem / Functors

```
map      :: (a->b) -> [a] -> [b]
                    map f [x,y,z]  ≡  [f x, f y, f z]
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
           zipWith f [x,y,z] [a,b]  ≡  [f x a, f y b]
filter  :: (a -> Bool) -> [a] -> [a]
               filter (/=y) [x,y,z]  ≡  [x,z]
foldr   :: (a -> b -> b) -> b -> [a] -> b
                   foldr f z [x,y]  ≡  x `f` (y `f` z)
foldl   :: (a -> b -> a) -> a -> [b] -> a
                   foldl f x [y,z]  ≡  (x `f` y) `f` z
```

### folds especiais

```
and :: [Bool] -> Bool         and [p,q,r]  ≡  p && q && r
or  :: [Bool] -> Bool          or [p,q,r]  ≡  p || q || r
sum     :: Num a => [a] -> a        sum [i,j,k]  ≡  i+j+k
product :: Num a => [a] -> a    product [i,j,k]  ≡  i*j*k
maximum :: Ord a => [a] -> a    maximum [9,0,5]  ≡  9
minimum :: Ord a => [a] -> a    minimum [9,0,5]  ≡  0
concat  :: [[a]] ->[a]    concat [xs,ys,zs]  ≡  xs++ys++zs
```

## Tuplas

```
fst     :: (a, b) -> a        fst (x,y)  ≡  x
snd     :: (a, b) -> b        snd (x,y)  ≡  y
curry   :: ((a, b) -> c) -> a -> b -> c
                   curry (\(x,y) -> e)  ≡  \x y -> e
uncurry :: (a -> b -> c) -> (a, b) -> c
                  uncurry (\x y -> e)  ≡  \(x,y) -> e
```

## Numéricas

```
abs       :: Num a => a -> a          abs (-9)  ≡  9
even, odd :: Integral a => a -> Bool   even 10  ≡  True
gcd, lcm :: Integral a => a -> a -> a  gcd 6 8  ≡  2
recip     :: Fractional a => a -> a    recip x  ≡  1/x
pi        :: Floating a => a                pi  ≡  3.14...
sqrt, log :: Floating a => a -> a       sqrt x  ≡  x**0.5
exp, sin, cos, tan, asin, acos :: Floating a => a -> a
truncate, round :: (RealFrac a, Integral b) => a -> b
ceiling, floor  :: (RealFrac a, Integral b) => a -> b
```

## Strings

```
lines   :: String -> [String]
                lines "ab\ncd\ne"  ≡  ["ab","cd","e"]
unlines :: [String] -> String
            unlines ["ab","cd","e"]  ≡  "ab\ncd\ne\n"
words   :: String -> [String]
                 words "ab cd e"  ≡  ["ab","cd","e"]
unwords :: [String] -> String
           unwords ["ab","cd","ef"]  ≡  "ab cd ef"
```

## Lê e Exibe classes

```
show :: Show a => a -> String    show 137  ≡  "137"
read :: Show a => String -> a    read "2"  ≡  2
```

## Classe Ord

```
min     :: Ord a => a -> a -> a     min 'a' 'b'  ≡  'a'
max     :: Ord a => a -> a -> a       max "bab"  ≡  "b"
compare :: Ord a => a->a->Ordering  compare 1 2  ≡  LT
```

## Bibliotecas / Módulos

| | |
|---|---|
| importando | `import Some.Module` |
| (qualificada) | `import qualified Some.Module as SM` |
| (subconj.) | `import Some.Module (foo,goo)` |
| (ocultação) | `import Some.Module hiding (foo,goo)` |
| (typeclass instances) | `import Some.Module ()` |
| declaração | `module Module.Name ( foo, goo ) where ...` |
| ./Arquivo/No/Disco.hs | `import Arquivo.No.Disco` |

## Rastreamento e monitoramento (inseguro) `Debug.Trace`

| | |
|---|---|
| Imprime *string*, return *expr* | `trace string $ expr` |
| Invoca show antes de impressão | `traceShow expr $ expr` |
| Função Trace | `f x y \| traceShow (x,y) False = undefined` |
| call values | `f x y = ...` |

## IO – Deve estar "dentro" de mônada IO

| | |
|---|---|
| Escreve char *c* para stdout | `putChar c` |
| Escreve string *cs* para stdout | `putStr cs` |
| Escreve string *cs* para stdout c/ novalinha | `putStrLn cs` |
| Imprime *x*, uma instância show, para stdout | `print x` |
| Lê char de stdin | `getChar` |
| Lê line de stdin como string | `getLine` |
| Lê toda entrada de stdin como string | `getContents` |
| Associa stdin/out a *foo* (*:: String -> String*) | `interact foo` |
| Escreve string *cs* em arquivo *fn* | `writeFile fn cs` |
| Acrescenta string *cs* para arquivo *fn* | `appendFile fn cs` |
| Lê conteúdo de arquivo *fn* | `readFile fn` |

## List Comprehensions

Pegue *pat* de *list*. Se *boolPredicate*, adicione elemento *expr* à lista:

```
[expr | pat <- list, boolPredicate, ...]
```

```
[x | x <- xs]              ≡  xs
[f x | x <- xs, p x]       ≡  map f $ filter p xs
[x | x <- xs, p x, q x]    ≡  filter q $ filter p xs
[x+y | x <- [a,b], y <- [i,j]] ≡ [a+i, a+j, b+i, b+j]
[x | boolE]                ≡  if boolE then [x] else []
```

## GHC - Glasgow Haskell Compiler (and Cabal)

| | |
|---|---|
| compilando program.hs | `$ ghc program.hs` |
| executando | `$ ./program` |
| executando diretamente | `$ run_haskell program.hs` |
| modo interativo (GHCi) | `$ ghci` |
| carga de prog no GHCi | `> :l program.hs` |
| recarga de prog no GHCi | `> :r` |
| ativa estatísticas GHCi | `> :set +s` |
| ajuda GHCi | `> :?` |
| Tipo de uma expressão | `> :t expr` |
| Info (oper./func./class) | `> :i thing` |
| Pacotes GHC instalados | `$ ghc-pkg list [pkg_name]` |
| Ativação de pragma | `{-# LANGUAGE Pragma #-}` |
| Idem, por chamada GHC | `$ ghc -XSomePragma ...` |
| instalada pacote pkg | `$ cabal install pkg` |
| atualiza lista de pacotes | `$ cabal update` |
| pacotes casando com pat | `$ cabal list pat` |
| informação sobre pacote | `$ cabal info pkg` |
| ajuda em comando | `$ cabal help [command]` |
| executa executável/test/bench | `$ cabal run/test/bench [name]` |
| inicializa sandbox | `$ cabal sandbox init` |
| adiciona fonte em sandbox | `$ cabal sandbox add-source dir` |