Existe ainda o tipo Vazio, `Void`, que vai ser importante aquando da definição de funções parciais. O único elemento deste tipo, $\perp$, é interpretado como o valor *indefinido*, permitindo deste modo a definição de funções que são indefinidas para certa classe de argumentos. É de notar que, em consequência do que se acabou de dizer, o elemento $\perp$ vai pertencer a todos os tipos.

Vejamos cada um destes tipos mais em pormenor:

Convenção Sintáctica: Os identificadores dos elementos começam por uma letra minúscula, os identificadores de tipos, e os identificadores dos construtores começam por uma letra maiúscula.

2.1.1 Tipo Unitário

O tipo unitário, `Unit`, é uma implementação do conjunto **1**.

Tipo	`()`
Valores	`()`

2.1.2 Bool

Valores Lógicos e símbolos proposicionais para uma lógica proposicional bivalente.

Tipo	`Bool`
Valores	`True`; `False`; `otherwise` ($\doteq$ True)
Operadores	`&&` (conjunção); $\|\|$ (disjunção); `not` (negação)
Predicados	$==, /=, <, <=, >=, >$

Os identificadores `True`, e `False` correspondem aos dois construtores 0-ários do conjunto `Bool`. O identificador `otherwise`, cujo valor é igual a `True` vai ser útil aquando da definição de funções com ramos.

2.1.3 Char

Caracteres, símbolos do alfabeto "Unicode".

Tipo	`Char`
Valores	símbolos da *Tabela Unicode* entre plicas, por exemplo `'a'`
Operadores	

A exemplo de outras linguagens o *Haskell* possui um conjunto de caracteres especiais: '`\t`', espaço tabular (tab); '`\n`', mudança de linha (newline); '`\\`' barra inclinada para trás, (backslash); '`\'`', plica; '`\"` ', aspas.

Tem-se acesso às duas funções de transferência usuais entre valores do tipo `Char` e valores do tipo `Int`, estas funções designam-se por: `ord :: Char -> Int` e `chr :: Int -> Char`.

Além destas funções temos também acesso às funções de transferência definidas para todo e qualquer tipo enumerado. No caso do tipo `Char` estas funções têm valores inteiros entre 0 e $2^{16} - 1$ inclusivé:

`toEnum` (Int $\rightarrow$ Char), por exemplo `(toEnum 97)::Char = 'a'`;

`fromEnum` (Char $\rightarrow$ Int), por exemplo `fromEnum 'a' = 97`.

Na expressão `(toEnum 97)::Char = 'a'`, a componente `::Char` serve para retirar a ambuiguidade que existe quanto ao co-domínio da função de transferência `toEnum`, isto dado que o co-domínio é dado pela classe de todos os tipos enumeráveis, mais à frente precisar-se-á o que se entende por classe de tipos. Esta função é um exemplo de uma função polimórfica (veremos isso mais à frente) sendo por tal possível aplica-la em diferentes situações.

2.1.4 Int, Integer, Float, Double

Valores numéricos, inteiros (`Int, Integer`), e reais (`Float, Double`).

Int, Integer

Inteiros.

Tipos	`Int, Integer`
Valores	`Int`, inteiros de comprimento fixo. A gama é de, pelo menos, -2^{29} a $2^{29}-1$. `Integer`, inteiros de comprimento arbitrário.
Operadores	`+, *, -, negate` ($\doteq$ `-` unário), `quot, div, rem, mod`

A gama de variação do tipo `Int` é nos dada pelas constantes `primMinInt`, e `primMaxInt`.

O operador `quot` faz o arredondamento "em direcção ao $-\infty$". O operador `div` faz o arredondamento "em direcção ao 0". Estes operadores verificam as seguintes equações:

```
(x `quot` y)*y+(x `rem` y) == x
(x `div` y)*y+(x `mod` y)  == x
```

A escrita de uma função binária entre plicas (acento grave), por exemplo `` `quot` `` permite a sua utilização como um operador infixo.

Funções de transferência: `even` (par), e `odd` (impar).

Float, Double

Reais.

Tipos	`Float, Double`
Valores	`Float`, reais precisão simples. `Double`, reais precisão dupla.
Operadores	`+, *, /, -, negate, ^, pi, exp, log, sqrt, **, logBase, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh.`

Em que `x^n` dá-nos a potência inteira de x, `x**y` dá-nos a exponencial de base x de y, e `logBase x y` dá-nos o logaritmo de base x de y.

Comum a todos os tipos numéricos temos os operadores `abs` e `signum`. Em que `signum(x)` é igual a -1, 0 ou 1, consoante x seja negativo, nulo ou positivo, respectivamente.

Funções de transferência: `ceiling, floor, truncate, round`. A função `ceiling(x)` dá-nos o menor inteiro que contêm x, `floor(x)` dá-nos o maior inteiro contido em x.

2.1.5 Tipo Vazio

Tipo	`Void`
Valores	$\perp$

Dado que o *Haskell* é uma linguagem não-estrita (veremos mais à frente o significado preciso desta afirmação) o valor $\perp$ (indefinido) está presente em todos os tipos.

2.2 Tipos Estruturados

O *Haskell* possui um conjunto de construtores de tipos que possibilitam a construção de novos tipos a partir dos tipos de base. Temos assim a possibilidade de construir novos tipos a partir do co-produto, do produto, e da exponenciação de conjuntos.

2.2.1 Produtos, (N-uplos)

O tipo correspondente a um dado n-uplo implementa o produto cartesiano dos tipos que compõem o n-uplo.

$$T = T_1 \times T_2 \times \cdots \times T_m$$

Tipo	`(`$\mathtt{T}_1$`,...,`$\mathtt{T}_n$`)`
Valores	`(`$\mathtt{e}_1$`,...,`$\mathtt{e}_k$`)`, com $k \geq 2$
Construtores	`(,...,)`
Operadores	`fst, snd`, só aplicáveis a pares.

Os operadores `fst(x,y) = x`, e `snd(x,y)=y` dão-nos as duas projecções próprias do produto cartesiano.

2.2.2 Co-produto (Enumeração)

O *Haskell* possuí também tipos definidos por enumeração, podemos por exemplo considerar que os tipos simples podem ser definidos desse modo. A definição de tipos por enumeração, e a utilização dos contructores próprios dos co-produtos irá ser explorada mais à frente aquando do estudo da definição de novos tipos em *Haskell*.

2.2.3 Listas

O conjunto das sequências finitas, eventualmente vazias, de elementos de um dado tipo A, isto é, $A^* = A^0 + A^1 + A^2 + \cdots + A^n$, é implementado através do tipo listas homogéneas.

Para um dado tipo de base `A` temos:

$$\text{Lista } \mathtt{A} = \text{Vazia} + \mathtt{A} \times \text{Lista } \mathtt{A}$$

Tipo	`[A]`, listas de elementos de um tipo `A`
Valores	`[]` (lista vazia); $[e_0,\ldots,e_m]$ (lista não vazia)
Construtores	`[]` : 1 $\longrightarrow$ Lista $*$ $\longmapsto$ `[]` : : A $\times$ Lista A $\longrightarrow$ Lista A (a,l) $\longmapsto$ l′ = a:l $[e_0,e_1,\ldots,e_m] \doteq e_0:(e_1:(\ldots:(e_m : []))\ldots)$
Operadores	`++, head, last, tail, init, nul, length, !!, foldl, foldl1, scanl, scanl1, foldr, foldr1, scanr, scanr1, iterate, repeat, replicate, cycle, take, drop, splitAt, takeWhile, dropWhile, span, break, lines, words, mlines, mwords, reverse, and, or, any, all, elem, notElem, lookup, sum, product, maximun, minimun, concatMap, zip, zip3, zipWith, zipWith3, unzip, unzip3`

De entre de todos estes operadores vejamos a definição de `last` e de `init`:

$last([e_0,e_1,\ldots,e_m]) = e_m$

$init([e_0,e_1,\ldots,e_m]) = [e_0,e_1,\ldots,e_{m-1}]$

O operador `!!` permite usar uma lista como uma tabela uni-dimensional com índices a começar no zero. Por exemplo:

$[e_0,e_1,\ldots,e_m]$ `!!` $k = e_k,\ 0 \leq k \leq m$

O operador `++` dá-nos a concatenação de duas listas. Por exemplo:

`[1,2,3]++[4,5,6] = [1,2,3,4,5,6]`

O *Haskell* possuí duas formas de construção de listas sem que para tal seja necessário escrever a lista com todos os seus elementos. Vejamos de seguida esses dois mecanismos.

Definição de uma lista através de uma gama de variação. Podemos definir uma lista de elemento numéricos através dos seus limites, ou ainda através dos seus limites e do incremento entre valores. Vejamos esses dois casos:

- `[n..m]` é a lista `[n,n+1,n+2,...,m]`. Se o valor de `n` exceder o de `m`, então a lista é vazia. Por exemplo:

```
[2..5] = [2,3,4,5]
[3.1..7.0] = [3.1,4.1,5.1,6.1]
[7..3] = []
```

- `[n,p..m]` é a lista de todos os valores desde `n` até `m`, em passos de `p-n`. Por exemplo:

```
[7,6..3] = [7,6,5,4,3]
[0.0,0.4..1.0] = [0.0,0.4,0.8]
```

Em ambos os caso o último elemento da lista é o último elemento a verificar a condição requerida, sem que o seu valor ultrapasse o valor limite.

Definição de uma lista por compreensão. O *Haskell* permite a construção de lista através de uma definição por compreensão, à imagem daquilo que estamos habituados em relação à definição de conjuntos. Pegando nessa analogia vejamos como se procede à definição de listas por compreensão.

Para conjuntos temos:

$$\{E(x) \mid x \in C \wedge P_1(x) \wedge \ldots \wedge P_n(x)\}$$

com $E(x)$ uma expressão em x, C um conjunto que é suposto conter x, e os vários $P_i(x)$ são proposições em x.

Para listas em *Haskell* tem-se:

```
[E(x) | x <- l, P1(x), ..., Pn(x)]
```

com `l` uma lista.

Por exemplo: `[2*a | a <- [2,4,7], even a, a>3]` dá-nos a lista de todos os `2*a` em que `a` pertence à lista `[2,4,7]`, e `a` é par, e `a` é maior do que 3, ou seja a lista `[8]`.

As sequências de proposições $P_i(x)$ pode ser vazia, por exemplo: `[2*a | a <- l]` é uma forma expedita de obter uma lista cujos elementos são em valor o dobro dos elementos de uma dada lista `l`.

2.2.4 Sequências de Caracteres ("Strings")

As sequências de caracteres são suficientemente importantes para terem um tratamento especial, no entanto elas podem sempre ser consideradas como uma lista de elementos do tipo `char`.

Tipo	String
Valores	sequências de caracteres entre aspas. Por exemplo `"abc"` ($\doteq$ ['a','b','c'])
Operadores	Os operadores das listas.

2.2.5 Funções

As funções vão estar subjacentes a todas as manipulações da linguagem e implementam a exponenciação de conjuntos. Ou seja o conjunto de todas as funções de A para B é dado por B^A.

Tipo	`a -> b`
Valores	`id`, `const`, ...
Construtores	`\ x -> exp(x)`
Operadores	$\cdot$, `curry`, `uncurry`, ...

O operador "$\cdot$" dá-nos a composição de funções. Os operadores `curry` e `uncurry` verificam, `curry f x y = f (x,y)` e `uncurry f p = f (fst p) (snd p)`.

A abstracção lambda permite-nos definir uma função sem lhe dar um nome. O construtor `->` permite-nos definir uma função atribuindo-lhe um nome.

2.3 Expressões

A linguagem *Haskell* é uma linguagem com uma disciplina de tipos rigorosa, quer isto dizer que qualquer expressão em *Haskell* tem um e um só tipo bem definido.

A este propósito é de referir uma série de comandos que se revelam interessantes; um deles é o comando `:type` (ou `:t`). Este comando dá-nos, para uma determinada expressão, a informação acerca do tipo da expressão. Por exemplo:

```
Prelude> :t 'a'
'a' :: Char
```

A sintaxe <*valor*>`::`<*tipo*> tem o significado óbvio de <*valor*>$\in$<*tipo*>.

Outro comando útil, é o comando `:set +t`, ou melhor a aplicação do comando `:set` ao valor `+t`. Este comando modifica o comportamento do interpretador de modo a que este passa a explicitar (`+`) a informação do tipo (`t`) da expressão que é sujeita a avaliação. A situação inversa, valor por omissão, é conseguida através do comando `:set -t`. Por exemplo:

```
Prelude> 7+3
10
Prelude> :set +t
Prelude> 7+3
10 :: Int
```

A avaliação de uma expressão é feita por redução da expressão à sua forma normal. Podemos modificar o comportamento do interpretador de forma a que este explicite dados estatísticos (`:set +s`) acerca das reduções necessárias para a avaliação da expressão. Por exemplo

```
Prelude> :set +s
Prelude> 7+3
10 :: Int
(10 reductions, 10 cells)
```

Posto isto a escrita de expressões e a sua avaliação segue as regras usuais. Por exemplo:

```
Prelude> "Haskell"
"Haskell" :: String
Prelude> 5+3*10
35 :: Int
Prelude> head [3,56,79]
3 :: Int
Prelude> fst ('a',1)
'a' :: Char
Prelude> 4*1.0
4.0 :: Double
```

2.3.1 Precedência e Associatividade

A precedência dos vários operadores assim como o modo como se procede a sua associação é dada na seguinte tabela.

Precedência	associa à esquerda	associa à direita	não associativo
9	!!	.	—
8	—	^,^^,**	—
7	*,/,`div`,`mod`, `rem`,`quot`	—	—
6	+,-	—	—
5	—	:,++	\\
4	—	—	==,/=,<,<=,>,>=, `elem`,`notElem`
3	—	&&	—
2	—	||	—
1	>>,>>=	—	—
0	—	$,`seq`	—