

## Na importação de um módulo por outro módulo:

- é possível fazer a importação de todas as entidades exportadas pelo módulo fazendo

```
import Nome_do_módulo
```

- é possível indicar explicitamente as entidades que queremos importar, fazendo

```
import Nome_do_módulo (entidades a importar)
```

- é possível indicar selectivamente as entidades que não queremos importar (importa-se tudo o que é exportado pelo outro módulo excepto o indicado)

```
import Nome_do_módulo hiding (entidades a não importar)
```

- é possível fazer com que as entidades importadas sejam referenciadas indicando o módulo de onde provêm como prefixo (seguido de '.') fazendo

```
import qualified Nome_do_módulo (entidades a importar)
```

(Pode ser útil para evitar *colisões* de nomes, pois é ilegal importar entidades diferentes que tenham o mesmo nome. Mas se for o mesmo objecto que é importado de diferentes módulos, não há colisão. Uma entidade pode ser importada via diferentes caminhos sem que haja conflitos de nomes.)

169

```
module Arvores(ArvBin(Vazia,Nodo), naArv, soma, mult) where

data ArvBin a = Vazia
              | Nodo a (ArvBin a) (ArvBin a)
              deriving Show

conta Vazia = 0
conta (Nodo _ e d) = 1 + (conta e) + (conta d)

soma Vazia = 0
soma (Nodo x e d) = x + (soma e) + (soma d)

mult Vazia = 1
mult (Nodo x e d) = x * (mult e) * (mult d)

naArv :: (Eq a) => a -> ArvBin a -> Bool
naArv _ Vazia = False
naArv x (Nodo y e d) | x==y      = True
                    | otherwise = (naArv x e) || (naArv x d)
```

171

## Um exemplo com módulos

Considere os módulos: [Listas](#), [Arvores](#), [Tempo](#), [Horas](#) e [Main](#), que pretendem ilustrar as diferentes formas de exportar e importar entidades.

```
module Listas where

soma [] = 0
soma (x:xs) = x + (soma xs)

conta = length

naLista x [] = False
naLista x (y:ys) = if x==y then True
                  else naLista x ys

mult = product

cauda (_,xs) = xs
```

170

```
module Tempo(Time, horas, minutos, meioDia, cauda) where

import Listas

data Time = Am Int Int
          | Pm Int Int
          | Total Int Int deriving Show

hValida (Total h m) = 0<=h && h<24 && 0<=m && m<60
hValida (Am h m)    = 0<=h && h<12 && 0<=m && m<60
hValida (Pm h m)    = 0<=h && h<12 && 0<=m && m<60

horas (Am h m)      = h
horas (Pm h m)      = h + 12
horas (Total h m)   = h

minutos (Am h m)     = m
minutos (Pm h m)     = m
minutos (Total h m)  = m

meioDia = (Total 12 00)

ex = cauda "experiencia"
```

172

```

module Horas(Hora(..), Tempo(manha)) where

data Hora = AM Int Int
          | PM Int Int

class Tempo a where
  manha :: a -> Bool
  tarde :: a -> Bool
  tarde t = not (manha t)

instance Tempo Hora where
  manha (AM _ _) = True
  manha (PM _ _) = False

```

173

Após carregar o módulo `Main`, analise o comportamento do interpretador.

```

*Main> soma arv1
15
*Main> mult arv1
Variable not in scope: `mult'
*Main> conta arv1
Variable not in scope: `conta'
*Main> Listas.soma lis1
10
*Main> mult lis1
Variable not in scope: `mult'
*Main> Listas.mult lis1
24

```

```

*Main> testeC
[2,3,4]
*Main> hValida meioDia
Variable not in scope: `hValida'

```

```

*Main> isDigit 'e'
Variable not in scope: `isDigit'
*Main> isAlpha 'e'
True
*Main> toUpper arv1
Nodo 25 (Nodo 9 Vazia (Nodo 16 Vazia Vazia))
(Nodo 4 (Nodo 1 Vazia Vazia) Vazia)
*Main> test
"testando"

```

```

*Main> minTotal meioDia
720
*Main> minTotal (Am 9 30)
Data constructor not in scope: `Am'
*Main> manha (AM 9 30)
True
*Main> tarde (PM 17 15)
Variable not in scope: `tarde'

```

175

```

module Main where

import Arvores (ArvBin(..), soma, naArv)
import qualified Listas (soma, mult, conta)
import Tempo
import Horas
import Char hiding (toUpper, isDigit)

arv1 = Nodo 5 (Nodo 3 Vazia (Nodo 4 Vazia Vazia))
        (Nodo 2 (Nodo 1 Vazia Vazia) Vazia)

lis1 = [1,2,3,4]

minTotal :: Time -> Int
minTotal t = (horas t)*60 + (minutos t)

testeC = cauda lis1

toUpper :: Num a => ArvBin a -> ArvBin a
toUpper Vazia = Vazia
toUpper (Nodo x e d) = Nodo (x*x) (toUpper e) (toUpper d)

test = map toLower "tesTAnDo"

```

174

## Compilação de programas Haskell

Para criar programas *executáveis* o compilador Haskell precisa de ter definido um módulo `Main` com uma função `main` que tem que ser de tipo `IO`.

A função `main` é o ponto de entrada no programa, pois é ela que é invocada quando o programa compilado é executado.

A compilação de um programa Haskell, usando o *Glasgow Haskell Compiler*, pode ser feita executando na shell do sistema operativo o seguinte comando:

```
ghc -o nome_do_executável --make nome_do_ficheiro_do_módulo_principal
```

**Exemplo:** Usando o último exemplo para testar a compilação de programas definidos em vários módulos, podemos acrescentar ao módulo `Main` a declaração

```
main = putStrLn "OK"
```

Assumindo que este módulo está guardado no ficheiro `Main.hs` podemos fazer a compilação assim:

```
ghc -o testar --make Main
```

**Exemplo:** Assumindo que o módulo do próximo slide está no ficheiro `roots.hs`, podemos gerar um executável (chamado `raizes`) fazendo

```
ghc -o raizes --make roots
```

176

```

module Main where

main :: IO ()
main = do calcRoots
    putStrLn "Deseja continuar (s/n) ? "
    x <- getLine
    case (head x) of
        's' -> main
        'S' -> main
        _   -> putStrLn "\n FIM."

calcRoots :: IO ()
calcRoots = do putStrLn "Calculo das raizes do polimomio a x^2 + b x + c"
    putStrLn "Indique o valor do ceoficiente a: "
    a1 <- getLine >>= readIO
    putStrLn "Indique o valor do ceoficiente b: "
    b1 <- getLine >>= readIO
    putStrLn "Indique o valor do ceoficiente c: "
    c1 <- getLine >>= readIO
    case (roots (a1,b1,c1)) of
        Nothing -> putStrLn "Nao ha' raizes reais"
        (Just (r1,r2)) -> putStrLn ("As raizes do polinomio sao "++
            (show r1)++" e "++(show r2))

roots :: (Float,Float,Float) -> Maybe (Float,Float)
roots (a,b,c)
    | d >= 0 = Just ((-b + (sqrt d))/(2*a), (-b - (sqrt d))/(2*a))
    | d < 0 = Nothing
    where d = b^2 - 4*a*c

```

177

## Tipos Abstractos de Dados

As assinaturas das funções do tipo abstracto de dados e as suas especificações constituem o **interface** do tipo abstracto de dados. Nem a estrutura interna do tipo abstracto de dados, nem a implementação destas funções são visíveis para o utilizador.

Dada a especificação de um tipo abstracto de dados, as operações que o definem poderão ter **diferentes implementações**, dependendo da estrutura usada na representação interna de dados e dos algoritmos usados.

A utilização de tipos abstractos de dados traz benefícios em termos de **modularidade** dos programas. Alterações na implementação das operações do tipo abstracto não afecta outras partes do programa desde que as operações mantenham o seu tipo e a sua especificação.

Em Haskell, a construção de tipos abstractos de dados é feita utilizando **módulos**.

O módulo onde se implementa o tipo abstracto de dados deve exportar apenas o nome do tipo e o nome das operações que constituem o seu interface. A representação do tipo fica assim escondida dentro do módulo, não sendo visível do seu exterior.

Deste modo, podemos mais tarde alterar a representação do tipo abstracto sem afectar os programas que utilizam esse tipo abstracto.

179

## Tipos Abstractos de Dados

A quase totalidade dos tipos de dados que vimos até aqui são **tipos concretos de dados**, dado que se referem a uma estrutura de dados concreta fornecida pela linguagem.

**Exemplos:**

```

data ArvBin a = Vazia
              | Nodo a (ArvBin a) (ArvBin a)

type TB = [(Integer,String)]

```

(ArvBin a) e TB são dois tipos concretos. Sabemos como são constituídos os valores destes tipos e podemos extrair informação ou construir novos valores, por manipulação directa dos construtores de valores destes tipos.

Em contraste, os **tipos abstractos de dados** não estão ligados a nenhuma representação particular. Em vez disso, eles são definidos implicitamente através de um conjunto de operações utilizadas para os manipular.

**Exemplo:** O tipo (IO a) é um tipo abstracto de dados. Não sabemos de que forma são os valores deste tipo. Apenas conhecemos um conjunto de funções para os manipular.

178

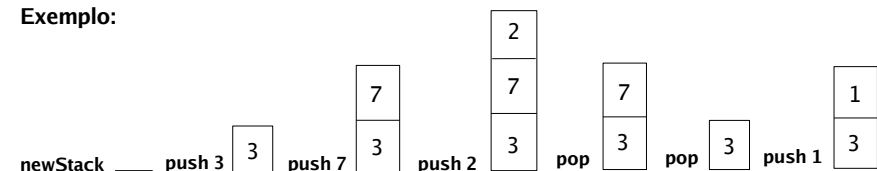
## Stacks (pilhas)

Uma **Stack** é uma colecção homogénea de itens que implementa a noção de **pilha**, de acordo com o seguinte interface:

<code>push :: a -&gt; Stack a -&gt; Stack a</code>	coloca um item no topo da pilha
<code>pop :: Stack a -&gt; Stack a</code>	remove o item do topo da pilha
<code>top :: Stack a -&gt; a</code>	dá o item que está no topo da pilha
<code>stackEmpty :: Stack a -&gt; Bool</code>	testa se a pilha está vazia
<code>newStack :: Stack a</code>	cria uma pilha vazia

Os itens da stack são removidos de acordo com a estratégia **LIFO** (Last In First Out).

**Exemplo:**



180

```

module Stack(Stack, push, pop, top, stackEmpty, newStack) where

push      :: a -> Stack a -> Stack a
pop       :: Stack a -> Stack a
top       :: Stack a -> a
stackEmpty :: Stack a -> Bool
newStack  :: Stack a

data Stack a = EmptyStk
             | Stk a (Stack a)

push x s = Stk x s

pop EmptyStk = error "pop em stack vazia."
pop (Stk _ s) = s

top EmptyStk = error "top em stack vazia."
top (Stk x _) = x

newStack = EmptyStk

stackEmpty EmptyStk = True
stackEmpty _       = False

instance (Show a) => Show (Stack a) where
    show (EmptyStk) = "#"
    show (Stk x s)  = (show x) ++ "|" ++ (show s)

```

181

```

module Stack(Stack, push, pop, top, stackEmpty, newStack) where

push      :: a -> Stack a -> Stack a
pop       :: Stack a -> Stack a
top       :: Stack a -> a
stackEmpty :: Stack a -> Bool
newStack  :: Stack a

data Stack a = Stk [a]

push x (Stk s) = Stk (x:s)

pop (Stk []) = error "pop em stack vazia."
pop (Stk (_:xs)) = Stk xs

top (Stk []) = error "top em stack vazia."
top (Stk (x:_)) = x

newStack = Stk []

stackEmpty (Stk []) = True
stackEmpty _       = False

instance (Show a) => Show (Stack a) where
    show (Stk []) = "#"
    show (Stk (x:xs)) = (show x) ++ "|" ++ (show (Stk xs))

```

183

```

module Main where

import Stack

listTostack :: [a] -> Stack a
listTostack [] = newStack
listTostack (x:xs) = push x (listTostack xs)

stackTolist :: Stack a -> [a]
stackTolist s
    | stackEmpty s = []
    | otherwise    = (top s):(stackTolist (pop s))

ex1 = push 2 (push 7 (push 3 newStack))
ex2 = push "abc" (push "xyz" newStack)

```

Exemplos:

```

*Main> ex1
2|7|3|#
*Main> ex2
"abc"|"xyz"|#

```

```

*Main> listTostack [1,2,3,4,5]
1|2|3|4|5|#
*Main> stackTolist ex2
["abc","xyz"]
*Main> stackTolist (listTostack [1,2,3,4,5])
[1,2,3,4,5]

```

182

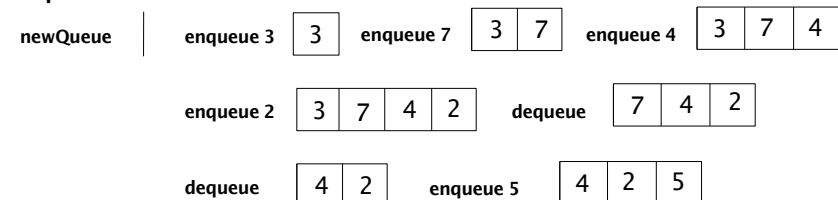
## Queues (filas)

Uma **Queue** é uma coleção homogênea de itens que implementa a noção de **fila de espera**, de acordo com o seguinte interface:

<code>enqueue :: a -&gt; Queue a -&gt; Queue a</code>	coloca um item no fim da fila de espera
<code>dequeue :: Queue a -&gt; Queue a</code>	remove o item do início da fila de espera
<code>front :: Queue a -&gt; a</code>	dá o item que está à frente na fila de espera
<code>queueEmpty :: Queue a -&gt; Bool</code>	testa se a fila de espera está vazia
<code>newQueue :: Queue a</code>	cria uma fila de espera vazia

Os itens da queue são removidos de acordo com a estratégia **FIFO (First In First Out)**.

Exemplo:



184

```

module Queue (Queue, enqueue, dequeue, front, queueEmpty, newQueue) where

enqueue    :: a -> Queue a -> Queue a
dequeue    :: Queue a -> Queue a
front      :: Queue a -> a
queueEmpty :: Queue a -> Bool
newQueue   :: Queue a

data Queue a = Q [a]

enqueue x (Q q) = Q (q++[x])

dequeue (Q (_:xs)) = Q xs
dequeue _         = error "Fila de espera vazia."

front (Q (x:_)) = x
front _         = error "Fila de espera vazia."

queueEmpty (Q []) = True
queueEmpty _      = False

newQueue = (Q [])

instance (Show a) => Show (Queue a) where
  show (Q []) = "."
  show (Q (x:xs)) = "<"++(show x)++(show (Q xs))

```

185

## Exemplos:

```

*Main> q1
<1<6<3.
*Main> queueTOSTack q1
3|6|1|#
*Main> invQueue q1
<3<6<1.

```

```

*Main> s1
2|8|9|#
*Main> stackTQueue s1
<2<8<9.
*Main> invStack s1
9|8|2|#

```

187

```

module Main where

import Stack
import Queue

queueTOSTack :: Queue a -> Stack a
queueTOSTack q = qts q newStack
  where qts q s
        | queueEmpty q = s
        | otherwise    = qts (dequeue q) (push (front q) s)

stackTQueue :: Stack a -> Queue a
stackTQueue s = stq s newQueue
  where stq s q
        | stackEmpty s = q
        | otherwise    = stq (pop s) (enqueue (top s) q)

invQueue :: Queue a -> Queue a
invQueue q = stackTQueue (queueTOSTack q)

invStack :: Stack a -> Stack a
invStack s = queueTOSTack (stackTQueue s)

q1 = enqueue 3 (enqueue 6 (enqueue 1 newQueue))
s1 = push 2 (push 8 (push 9 newStack))

```

186

## Sets (conjuntos)

Um **Set** é uma colecção homogénea de itens que implementa a noção de **conjunto**, de acordo com o seguinte interface:

<code>emptySet :: Set a</code>	cria um conjunto vazio
<code>setEmpty :: Set a -&gt; Bool</code>	testa se um conjunto é vazio
<code>inSet :: (Eq a) =&gt; a -&gt; Set a -&gt; Bool</code>	testa se um item pertence a um conjunto
<code>addSet :: (Eq a) =&gt; a -&gt; Set a -&gt; Set a</code>	acrescenta um item a um conjunto
<code>delSet :: (Eq a) =&gt; a -&gt; Set a -&gt; Set a</code>	remove um item de um conjunto
<code>pickSet :: Set a -&gt; a</code>	escolhe um item de um conjunto

É necessário testar a igualdade entre itens, por isso o tipo dos itens tem que pertencer à classe Eq. Mas certas implementações do tipo Set podem requerer outras restrições de classe sobre o tipo dos itens.

É possível estabelecer um interface mais rico para o tipo abstracto Set, por exemplo, incluindo operações de **união**, **intersecção** ou **diferença** de conjuntos, embora se consiga definir estas operações à custa do interface actual.

A seguir apresentam-se duas implementações para o tipo abstracto Set.

188

```

module Set(Set, emptySet, setEmpty, inSet, addSet, delSet) where
emptySet :: Set a
setEmpty :: Set a -> Bool
inSet    :: (Eq a) => a -> Set a -> Bool
addSet   :: (Eq a) => a -> Set a -> Set a
delSet   :: (Eq a) => a -> Set a -> Set a
pickSet  :: Set a -> a

data Set a = S [a]    -- listas com repetições

emptySet = S []

setEmpty (S []) = True
setEmpty _      = False

inSet _ (S [])      = False
inSet x (S (y:ys)) | x == y = True
                  | otherwise = inSet x (S ys)

addSet x (S s) = S (x:s)

delSet x (S s) = S (delete x s)

delete x [] = []
delete x (y:ys) | x == y = delete x ys
                | otherwise = y:(delete x ys)

pickSet (S []) = error "Conjunto vazio"
pickSet (S (x:_)) = x

```

189

## Tables (tabelas)

(Table a b) é uma colecção de associações entre *chaves* do tipo *a* e *valores* do tipo *b*, implementando assim uma função finita, com domínio em *a* e co-domínio em *b*, através de uma determinada estrutura de dados.

O tipo abstracto *tabela* poderá ter o seguinte interface:

```

newTable :: Table a b

findTable :: (Ord a) => a -> Table a b -> Maybe b

updateTable :: (Ord a) => (a,b) -> Table a b -> Table a b

removeTable :: (Ord a) => a -> Table a b -> Table a b

```

Para permitir implementações eficientes destas operações, está-se a exigir que o tipo das chaves pertença à classe *Ord*.

A seguir apresentam-se duas implementações distintas para o tipo abstracto *tabela*:

- usando uma lista de pares (*chave,valor*) ordenada por ordem crescente das chaves;
- usando uma árvore binária de procura com pares (*chave, valor*) nos nodos da árvore.

191

```

module Set(Set, emptySet, setEmpty, inSet, addSet, delSet) where
emptySet :: Set a
setEmpty :: Set a -> Bool
inSet    :: (Eq a) => a -> Set a -> Bool
addSet   :: (Eq a) => a -> Set a -> Set a
delSet   :: (Eq a) => a -> Set a -> Set a
pickSet  :: Set a -> a

data Set a = S [a]    -- listas sem repetições

emptySet = S []

setEmpty (S []) = True
setEmpty _      = False

inSet _ (S [])      = False
inSet x (S (y:ys)) | x == y = True
                  | otherwise = inSet x (S ys)

addSet x (S s) | (elem x s) = S s
                | otherwise = S (x:s)

delSet x (S s) = S (delete x s)

delete x [] = []
delete x (y:ys) | x == y = ys
                | otherwise = y:(delete x ys)

pickSet (S []) = error "Conjunto vazio"
pickSet (S (x:_)) = x

```

190

```

module Table(Table, newTable, findTable, updateTable, removeTable) where

newTable    :: Table a b
findTable   :: (Ord a) => a -> Table a b -> Maybe b
updateTable :: (Ord a) => (a,b) -> Table a b -> Table a b
removeTable :: (Ord a) => a -> Table a b -> Table a b

data Table a b = Tab [(a,b)]    -- lista ordenada por ordem crescente

newTable = Tab []

findTable _ (Tab []) = Nothing
findTable x (Tab ((c,v):cvs))
    | x < c = Nothing
    | x == c = Just v
    | x > c = findTable x (Tab cvs)

updateTable (x,z) (Tab []) = Tab [(x,z)]
updateTable (x,z) (Tab ((c,v):cvs))
    | x < c = Tab ((x,z):(c,v):cvs)
    | x == c = Tab ((c,z):cvs)
    | x > c = let (Tab t) = updateTable (x,z) (Tab cvs)
               in Tab ((c,v):t)

```

{- -- continua -- -}

192

*{- -- continuação do slide anterior -- -}*

```
removeTable _ (Tab []) = Tab []
removeTable x (Tab ((c,v):cvs))
    | x < c = Tab ((c,v):cvs)
    | x == c = Tab cvs
    | x > c = let (Tab t) = removeTable x (Tab cvs)
              in Tab ((c,v):t)

instance (Show a, Show b) => Show (Table a b) where
    show (Tab []) = ""
    show (Tab ((c,v):cvs)) = (show c)++"\t"++(show v)++"\n"++(show (Tab cvs))
```

Evita-se derivar o método show de forma automática, para não revelar a implementação do tipo abstracto.

193

*{- -- continuação do slide anterior -- -}*

```
removeTable _ Empty = Empty
removeTable x (Node (c,_) e Empty) | x == c = e
removeTable x (Node (c,_) Empty d) | x == c = d
removeTable x (Node (c,v) e d)
    | x < c = Node (c,v) (removeTable x e) d
    | x > c = Node (c,v) e (removeTable x d)
    | x == c = let (y,z) = minTable d
               in Node (y,z) e (removeTable y d)

minTable :: Table a b -> (a,b)
minTable (Node (c,v) Empty _) = (c,v)
minTable (Node _ e _) = minTable e

instance (Show a, Show b) => Show (Table a b) where
    show Empty = ""
    show (Node (c,v) e d) = (show e)++(show c)++"\t"++(show v)++"\n"++(show d)
```

195

```
module Table (Table, newTable, findTable, updateTable, removeTable) where

newTable :: Table a b
findTable :: (Ord a) => a -> Table a b -> Maybe b
updateTable :: (Ord a) => (a,b) -> Table a b -> Table a b
removeTable :: (Ord a) => a -> Table a b -> Table a b

-- Arvore binaria de procura
data Table a b = Empty
               | Node (a,b) (Table a b) (Table a b)

newTable = Empty

findTable _ Empty = Nothing
findTable x (Node (c,v) e d)
    | x < c = findTable x e
    | x == c = Just v
    | x > c = findTable x d

updateTable (x,z) Empty = Node (x,z) Empty Empty
updateTable (x,z) (Node (c,v) e d)
    | x < c = Node (c,v) (updateTable (x,z) e) d
    | x == c = Node (c,z) e d
    | x > c = Node (c,v) e (updateTable (x,z) d)
```

*{- -- continua -- -}*

194

module Main where

import Table

```
type Numero = Integer
type Nome = String
type Nota = Integer
```

```
pauta :: [(Numero,Nome,Nota)] -> Table Numero (Nome,Nota)
pauta [] = newTable
pauta ((x,y,z):xyzs) = updateTable (x,(y,z)) (pauta xyzs)
```

```
info = [(1111,"Mario",14), (5555,"Helena",15), (3333,"Teresa",12),
        (7777,"Pedro",15), (2222,"Rui",17), (9999,"Pedro",10)]
```

Exemplos:

```
*Main> pauta info
1111 ("Mario",14)
2222 ("Rui",17)
3333 ("Teresa",12)
5555 ("Helena",15)
7777 ("Pedro",15)
9999 ("Pedro",10)
```

```
*Main> findTable 5555 (pauta info)
Just ("Helena",15)
*Main> findTable 8888 (pauta info)
Nothing
*Main> removeTable 9999 (pauta info)
1111 ("Mario",14)
2222 ("Rui",17)
3333 ("Teresa",12)
5555 ("Helena",15)
7777 ("Pedro",15)
```

Como estará a tabela implementada ?

196