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.

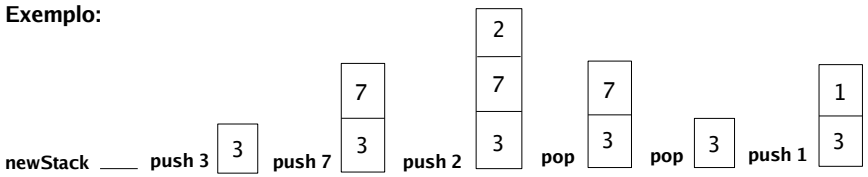
Stacks (pilhas)

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

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

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

Exemplo:




```haskell
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)
```



```haskell
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]
```

```haskell
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))
```

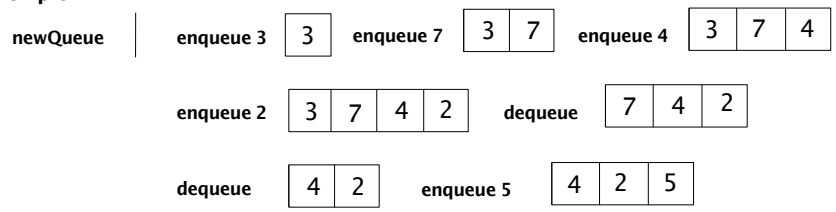

Queues (filas)

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

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

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

Exemplo:




```haskell
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))
```



```haskell
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)

stackTOqueue :: Stack a -> Queue a
stackTOqueue 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  = stackTOqueue (queueTOstack q)

invStack :: Stack a -> Stack a
invStack s  = queueTOstack (stackTOqueue s)

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

Exemplos:

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

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


Sets (conjuntos)

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

`emptySet :: Set a`	cria um conjunto vazio
`setEmpty :: Set a -> Bool`	testa se um conjunto é vazio
`inSet :: (Eq a) => a -> Set a -> Bool`	testa se um item pertence a um conjunto
`addSet :: (Eq a) => a -> Set a -> Set a`	acrescenta um item a um conjunto
`delSet :: (Eq a) => a -> Set a -> Set a`	remove um item de um conjunto
`pickSet :: Set a -> a`	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 establecer 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.


```
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
```



```
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
```

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.


```
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 -- -}

{- -- 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.


```
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 -- -}