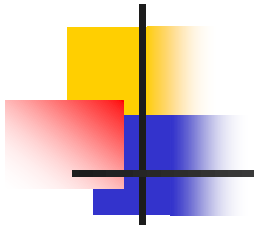
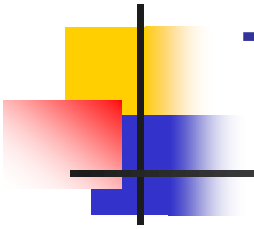


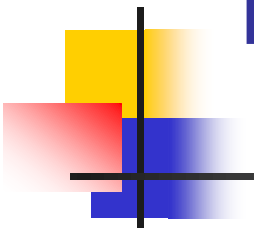
Tipos Abstratos de Dados

Introdução



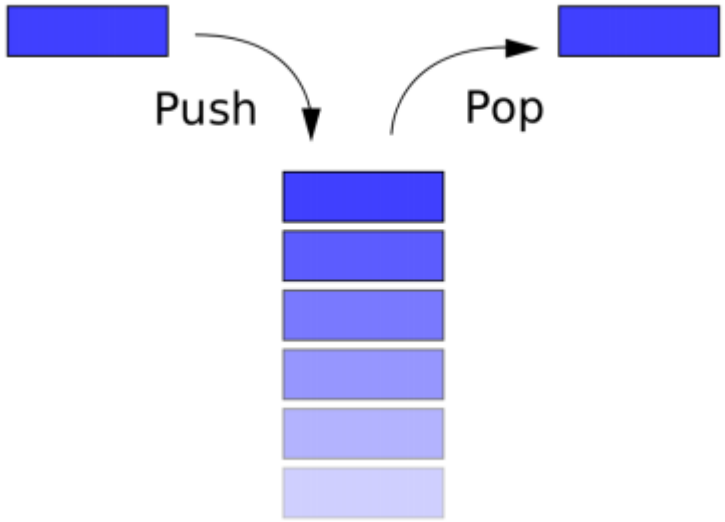
- Podemos começar por definir as operações que um tipo deve suportar, sem especificar a representação.
- Tipos definidos apenas pelas suas operações dizem-se abstratos.

Definindo o tipo Pilha



- Uma pilha é uma estrutura de dados que suporta as seguintes operações:
  - *push*: acrescenta um valor ao topo da pilha;
  - *pop*: remove o valor do topo da pilha;
  - *top*: obter o valor no topo da pilha;
  - *empty*: cria uma pilha vazia;
  - *isEmpty*: testa se uma pilha é vazia.

Definindo o tipo Pilha(cont.)



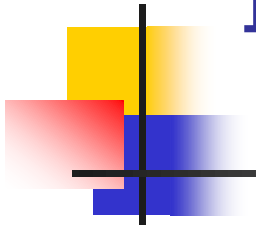
A pilha é uma estrutura LIFO (“last-in, first-out”): o último valor a ser colocado é o primeiro a ser removido.

Definindo o tipo Pilha (cont.)

- Em Haskell vamos representar pilhas por um tipo paramétrico `Stack` e uma função para cada operação.

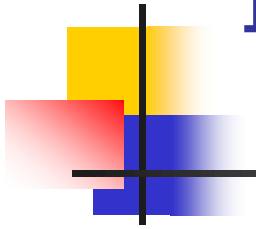
```
data Stack a     -- pilha com valores de tipo 'a'

push :: a -> Stack a -> Stack a
pop :: Stack a -> Stack a
top :: Stack a -> a
empty :: Stack a
isEmpty :: Stack a -> Bool
```

Implementando o tipo Pilha (cont.)



- Para implementar o tipo abstrato vamos:
  - Escolher uma representação concreta e implementar as operações.
  - Ocultar a representação concreta permitindo apenas usar as operações.
- Em Haskell realizamos estas operações usando a estrutura de **módulos**.

Implementando o tipo Pilha (cont.)



- Um módulo é um conjunto de definições relacionadas (tipos, constantes, funções. . .)
- Definimos um módulo `Foo` num arquivo `Foo.hs` com a declaração:

```
module Foo where
```

- Para usar o módulo `Foo` colocamos uma declaração

```
import Foo
```

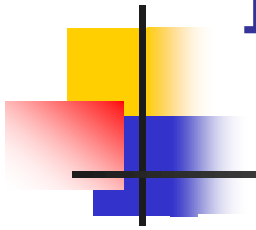
- Por omissão, todas as definições num módulo são exportadas; podemos restringir as entidades exportadas:

```
import Foo (T1,T2,f1,f2, ...) where
```

Implementando o tipo Pilha (cont.)

```
module Stack (Stack,  -- exportar o tipo mas não o construtor
              push, pop, top,    -- exportar as operações
              empty, isEmpty) where

data Stack a = Stk [a]        -- representação usando listas

push :: a -> Stack a -> Stack a
push x (Stk xs) = Stk (x:xs)

pop :: Stack a -> Stack a
pop (Stk (_:xs)) = Stk xs
pop _            = error "Stack.pop: empty stack"
```

Implementando o tipo Pilha (cont.)



```
top :: Stack a -> a
top (Stk (x:_)) = x
top _           = error "Stack.top: empty stack"

empty :: Stack a
empty = Stk []

isEmpty :: Stack a -> Bool
isEmpty (Stk [])= True
isEmpty (Stk _) = False
```

Usando o tipo Pilha

- Exemplo: deseja-se obter o número de elementos de uma pilha:

```
import Stack

size :: Stack a -> Int
size s | isEmpty s = 0
       | otherwise = 1 + size (pop s)
```

Esta função usa apenas as operações abstratas sobre pilhas, não a representação concreta.

Ocultamento da Informação

```
import Stack

size :: Stack a -> Int
size (Stk xs) = length xs                    -- ERRO
```

- Esta definição é rejeitada porque o construtor de pilhas `Stk` é invisível fora do módulo `Stack` (logo não podemos usar encaixe de padrões).
- Também não podemos construir pilhas usando `Stk` apenas de usar as operações `push` e `empty`.

Propriedades da Pilhas

- Podemos especificar o comportamento das operações dum tipo de dados abstrato usando equações algébricas.
- Exemplo: qualquer implementação de pilhas deve verificar as condições (1)–(4) para quaisquer valor x e pilha s.

$$pop\ (push\ x\ s) = s \qquad (1)$$

$$top\ (push\ x\ s) = x \qquad (2)$$

$$isEmpty\ empty = True \qquad (3)$$

$$isEmpty\ (push\ x\ s) = False \qquad (4)$$

Propriedades de uma pilha (cont.)

- Vamos verificar a propriedade (1) para a implementação com listas; temos s = Stk xs em que xs é uma lista.

$$
\begin{aligned}
& pop\ (push\ x\ \overbrace{(Stk\ xs)}^{s}) \\
= \quad & \{\text{pela definição de } push\} \\
& pop\ (Stk\ (x : xs)) \\
= \quad & \{\text{pela definição de } pop\} \\
& \underbrace{Stk\ xs}_{s}
\end{aligned}
$$