# Script: Abstract data types in Haskell

## Headline

Abstract data types in Haskell

## Description

We look at functional programming techniques for implementing abstract data types and concrete data structures. In particular, we want to get a basic understanding of using familiar data types such as stacks. This is also an exercise in modularity (or information hiding).

## Concepts

- Concrete data type
- Abstract data type
- Stack as a key example of an abstract data type
- Reverse Polish notation as an illustrative application of Stack
- Functions as data as an advanced technique

---

# Concept: Modularity

## Headline

A property of software designs to be based on separate and recombinable components

## Illustration

See the related terms modular programming and modularity.

---

# Concept: Abstract data type

## Headline

A data type that does not reveal representation

## Illustration

An abstract data type is usually just defined through the list of operations on the type, possibly enirched (formaly or informaly) by properties (invariants, pre- and postconditions).

Consider the following concrete data type for points:

```
data Point = Point { getX :: Int, getY :: Int }
  deriving (Eq, Show, Read)
```

Now suppose we want to hide the precise representation of points. In particular, we want to rule out that programmers can match and apply the constructor *Point*. The existing getters are sufficient to observe points without matching, but we need to provide some "public" means of constructing points.

```
mkPoint :: Int -> Int -> Point
mkPoint = Point
```

The idea is now to export *mkPoint*, but not the constructor, thereby making possible representation changes without changing any code that uses points. This is, of course, a trivial example, as the existing representation of points is probably quite appropriate, but see a more advanced illustration for an abstract data type Stack.

A complete module for an abstract data type for points may then look like this:

```
module Point(
  Point, -- constructor is NOT exported
  mkPoint,
  getX,
  getY
) where

data Point = Point { getX :: Int, getY :: Int }
  deriving (Eq, Show, Read)

mkPoint :: Int -> Int -> Point
mkPoint = Point
```

When defining an abstract data type, we take indeed the point of view that the representation and thus the implementation as such is not known or not to be looked at. Hence, ideally, the intended functionality should be described in some other way. For instance, we may describe the functionality by properties. For instance, in Haskell we may declare testable Technology:QuickCheck properties like this:

```
prop_getX :: Int -> Int -> Bool
prop_getX x y = getX (mkPoint x y) == x

prop_getY :: Int -> Int -> Bool
prop_getY x y = getY (mkPoint x y) == y
```

These properties describe the (trivial) correspondence between construction with *mkPoint* and observation with *getX* and *getY*. Logically, the first property says that for all given *x* and *y*, we can construct a point and we can retrieve *x* again from that point with *getX*.

## Relationships

- An abstract data type is the opposite of a concrete data type.
- An abstract data type performs information hiding.

---

# Concept: Concrete data type

## Headline

A data type defined in terms of a concrete representation

## Relationships

- Non-concrete data types are called abstract data types.
- The term "concrete data type" is similar to the term "data structure".

---

# Concept: Information hiding

## Headline

The principle of information hiding

## Illustration

See the abstract data type Stack for an illustration of information hiding such that different kinds of representations are exercised for stacks with more or less information hiding applied to the representation.

---

# Concept: Stack

## Headline

A last in, first out (LIFO) abstract data type

## Illustration

A simple implementation of stacks (of ints) is shown here as a functional data structure in Language:Haskell:.

```
{-| A simple implementation of stacks in Haskell -}

module Stack (
  Stack,
  empty,
  isEmpty,
  push,
  top,
  pop,
  size
) where

-- | Data structure for representation of stacks
data Stack = Empty | Push Int Stack

{- Operations on stacks -}

-- | Return the empty stack
empty :: Stack
empty = Empty

-- | Test for the empty stack
isEmpty :: Stack -> Bool
isEmpty Empty = True
isEmpty (Push _ _) = False

-- | Push an element onto the stack
push :: Int -> Stack -> Stack
push = Push

-- | Retrieve the top-of-stack, if available
top :: Stack -> Int
top (Push x s) = x

-- | Remove the top-of-stack, if available
pop :: Stack -> Stack
pop (Push x s) = s

-- | Compute size of stack
size :: Stack -> Int
size Empty = 0
size (Push _ s) = 1 + size s
```

These stacks are immutable. The push operation does not modify the given stack; it returns a new stack which shares the argument stack possibly with other parts of the program. The pop operation does not modify the given stack; it returns a part of the argument stack. We refer to Document:Handbook of data structures and applications for a profound discussion of functional data structures including the stack example. The functions for operations top and pop, as given above, are partial because they are undefined for the empty stack.

There are also alternative illustrative Stack implementations available:

https://github.com/101companies/101repo/tree/master/concepts/Stack

### Stacks as lists without information hiding

```
{-|

A leaky list-based implementation of stacks in Haskell.
```

```
That is, the representation type is not hidden.

-}

module LeakyListStack (
  Stack,
  empty,
  isEmpty,
  push,
  top,
  pop,
  size
) where

-- | Data structure for representation of stacks
type Stack = [Int]

{- Operations on stacks -}

-- | Return the empty stack
empty :: Stack
empty = []

-- | Test for the empty stack
isEmpty :: Stack -> Bool
isEmpty = null

-- | Push an element onto the stack
push :: Int -> Stack -> Stack
push = (:)

-- | Retrieve the top-of-stack, if available
top :: Stack -> Int
top = head

-- | Remove the top-of-stack, if available
pop :: Stack -> Stack
pop = tail

-- | Compute size of stack
size :: Stack -> Int
size = length
```

That is, stacks are represented as lists while the *Stack* type is simply defined as a type synonym to this end. This implementation does not enforce information hiding.

## Stacks as lists with information hiding

```
{-|

An opaque list-based implementation of stacks in Haskell.
That is, the representation type is hidden.

-}

module OpaqueListStack (
  Stack,
  empty,
  isEmpty,
  push,
  top,
  pop,
  size
) where

-- | Data structure for representation of stacks
newtype Stack = Stack { getStack :: [Int] }

{- Operations on stacks -}

-- | Return the empty stack
empty :: Stack
empty = Stack []

-- | Test for the empty stack
isEmpty :: Stack -> Bool
```

```
isEmpty = null . getStack

-- | Push an element onto the stack
push :: Int -> Stack -> Stack
push x s = Stack ( x : getStack s)

-- | Retrieve the top-of-stack, if available
top :: Stack -> Int
top = head . getStack

-- | Remove the top-of-stack, if available
pop :: Stack -> Stack
pop = Stack . tail . getStack

-- | Compute size of stack
size :: Stack -> Int
size = length . getStack
```

As before, stacks are represented as lists, but the *Stack* type is defined as a newtype which hides the representation as its constructor is not exported.

## Stack with length

```
{-|

An opaque list-based implementation of stacks in Haskell.
That is, the representation type is hidden.
The size of the stack is readily maintained.
Thus, the size can be returned with traversing the stack.

-}

module FastListStack (
  Stack,
  empty,
  isEmpty,
  push,
  top,
  pop,
  size
) where

-- | Data structure for representation of stacks
data Stack = Stack { getStack :: [Int], getSize :: Int }

{- Operations on stacks -}

-- | Return the empty stack
empty :: Stack
empty = Stack [] 0

-- | Test for the empty stack
isEmpty :: Stack -> Bool
isEmpty = null . getStack

-- | Push an element onto the stack
push :: Int -> Stack -> Stack
push x s
  = Stack {
      getStack = x : getStack s,
      getSize = getSize s + 1
    }

-- | Retrieve the top-of-stack, if available
top :: Stack -> Int
top = head . getStack

-- | Remove the top-of-stack, if available
pop :: Stack -> Stack
pop s
  = Stack {
      getStack = tail (getStack s),
      getSize = getSize s - 1
    }

-- | Compute size of stack
```

```
size :: Stack -> Int
size = getSize
```

As before, stacks are represented as lists and again this representation is hidden, but an additional data component for the size of the stack is maintained so that the size of a stack can be returned without traversing the stack.

## An application of stacks

See Concept:Reverse_Polish_notation.

---

# Concept: Functions as data

## Headline

The notion of functions being actual data

## Illustration

The notion of "functions as data" is closely related to the notion of higher-order functions. If one wishes to make a difference, then "functions as data" could be meant to focus on the aspect that functions may appear within data structures. The following illustration focuses on this aspect indeed.

Consider the following routine code for evaluating reverse polish notation:

```
-- Evaluation of RPN via stack
eval :: RPN -> Int
eval = loop empty
  where
    -- Loop over input
    loop :: Stack Int -> RPN -> Int
    loop s i =
      if null i
        then if size s == 1
               then top s
               else rpnError
        else
          loop (step (head i) s) (tail i)
```

Now let's assume that we want to parametrize this code in the stack *implementation*. Thus, we would need to pass a data structure to the eval function such that this structure essentially lists all the stack operations needed. The corresponding data structure can be set up as a record type like this:

```
data StackImpl s a =
  StackImpl {
    getEmpty :: s a,
    getPush :: a -> s a -> s a,
    getPop :: s a -> s a,
    getTop :: s a -> a,
    getSize :: s a -> Int
  }
```

A record for a specific stack implementation can be constructed like this:

```
import qualified SimpleStackADT as Simple

simpleImpl :: StackImpl Simple.Stack a
simpleImpl = StackImpl {
   getEmpty = Simple.empty,
   getPush = Simple.push,
   getPop = Simple.pop,
   getTop = Simple.top,
   getSize = Simple.size
 }
```

The parameterized version of the RPN evaluator looks like this:

```
-- Evaluation of RPN via stack
eval :: forall s. StackImpl s Int -> RPN -> Int
eval si = loop (getEmpty si)
  where
    -- Loop over input
    loop :: s Int -> RPN -> Int
    loop s i =
      if null i
        then if getSize si s == 1
               then getTop si s
               else rpnError
        else
```

```
    loop (step (head i) s) (tail i)
```

The parametrization boils down to this type:

```
StackImpl s Int
```

That is, the type variable *s* stands for the type constructor used by a specific stack implementation.

We need to pick a stack implementation when invoking the evaluator:

```
main = do
    print $ eval simpleImpl sample
```

---