COM2001: Abstract Data Types

1. Abstract Data Types

   A software designer has been asked to create a new data type called *Library* for a Haskell-based library-loan system.[1] The following algebraic data types have already been defined; they represent the library's books, readers and loans. For example, the loan `(Book 24, Reader 5)` represents the fact that the book with ID number 24 is currently on loan to the reader with ID 5.

```
data Book   = Book { bookID :: Int }
data Reader = Reader { readerID :: Int }
type Loan   = (Book, Reader)
```

The designer wants the data type *Library* to include functions that record

- the library's books (no two books can have the same ID);
- the library's readers (no two readers can have the same ID);
- whether or not a book is currently on loan, and if so, to which reader. A book can only be issued to a reader if
  - the book is listed in the library's collection;
  - the reader is a registered user of the library;
  - the book is not already on loan.

The required functions are:

Function	Required Behaviour
newLib	Create a new library with no books, readers or loans
getBooks	Given a library, return a list of all books in that library
getReaders	Given a library, return a list of all readers using that library
getLoans	Given a library, return a list of all current loans
addBook	Given a library and a book, add the book to the library and return the updated library
addReader	Given a library and a book, add the reader to the library and return the updated library
addLoan	Given a library, a book and a reader, issue the book to the reader and return the updated library
delBook	Given a library and a book, remove the book from the library's collection and return the updated library
delReader	Given a library and a reader, remove the reader from the library's list of registered users and return the updated library
delLoan	Given a library, a book and a reader, cancel the loan of the book to the reader, and return the updated library

a) Which of the functions might potentially generate error conditions? Explain the circumstances under which each error would occur. [10 marks]

ANSWER:

[1]NOTE: This specification is potentially ambiguous. You can resolve the ambiguities any way you like, but be sure to (a) say where you've done so, and (b) explain what you've done and why.

newLib	: no error conditions
getBooks	: no error conditions
getReaders	: no error conditions
getLoans	: no error conditions
addBook	: adding a book that is already present
addReader	: adding a reader who is already registered
addLoan	: lending a non-existent book; lending to an unregistered reader; lending a book that is already on loan;
delBook	: deleting a non-existent book
delReader	: deleting a non-existent reader
delLoan	: deleting a non-existent loan

b) Write down the syntax of the required ADT. [10 marks]

ANSWER:

I am assuming the existence of auxiliary types *ErrBook*, *ErrReader* and *ErrLoan*, representing errors relating to books, readers and loans, respectively. I am also assuming that the list constructor [ ] is defined.

newLib	: $\mathit{Library}$
getBooks	: $\mathit{Library} \rightarrow [\mathit{Book}]$
getReaders	: $\mathit{Library} \rightarrow [\mathit{Reader}]$
getLoans	: $\mathit{Library} \rightarrow [\mathit{Loan}]$
addBook	: $\mathit{Library} \rightarrow \mathit{Book} \rightarrow (\mathit{Library} \cup \mathit{ErrBook})$
addReader	: $\mathit{Library} \rightarrow \mathit{Reader} \rightarrow (\mathit{Library} \cup \mathit{ErrReader})$
addLoan	: $\mathit{Library} \rightarrow \mathit{Book} \rightarrow \mathit{Reader} \rightarrow (\mathit{Library} \cup \mathit{ErrBook} \cup \mathit{ErrReader} \cup \mathit{ErrLoan})$
delBook	: $\mathit{Library} \rightarrow \mathit{Book} \rightarrow (\mathit{Library} \cup \mathit{ErrBook})$
delReader	: $\mathit{Library} \rightarrow \mathit{Reader} \rightarrow (\mathit{Library} \cup \mathit{ErrReader})$
delLoan	: $\mathit{Library} \rightarrow \mathit{Book} \rightarrow \mathit{Reader} \rightarrow (\mathit{Library} \cup \mathit{ErrLoan})$

c) Write down the *Library* ADT's sorts, together with any specific values that need to be defined (for example, if one of the sorts is *Bool*, then required values of type *Bool* might be *true* and *false*). Explain what the sorts and values are used to represent [5 marks]

ANSWER:

Library	the type being defined
Book	the existing type used to represent books. We also need to represent lists of books, and assume the existence of an empty such list, *noBooks*.
Reader	the existing type used to represent readers. We also need to represent lists of readers, and assume the existence of an empty such list, *noReaders*.

Loan	the existing type used to represent loans. We also need to represent lists of loans, and assume the existence of an empty such list, *noLoans*. The Haskell declaration tells us that 3 functions are defined in relation to *Loan*, which we can call *fst* : *Loan* $\rightarrow$ *Book*; *snd* : *Loan* $\rightarrow$ *Reader*; and *mkPair* : *Book* $\rightarrow$ *Reader* $\rightarrow$ *Loan*, satisfying – $mkPair\ (fst\ l)\ (snd\ l) = l$ – $fst\ (mkPair\ b\ r) = b$ – $snd\ (mkPair\ b\ r) = r$
ErrBook	a sort representing errors relating to *Book*, including specific values: – *errNoSuchBook*– the specified book is not part of the library collection – *errDuplicateBook*– the specified book is already in the library
ErrReader	a sort representing errors relating to *Reader*, including specific values: – *errNoSuchReader*– the specified reader isn't registered – *errDuplicateReader*– the specified reader is already registered
ErrLoan	a sort representing errors relating to *Loan*, including specific values: – *errNoSuchLoan*– the specified loan doesn't exist – *errDuplicateLoan*– the specified book is already on loan

d) Which of *Library*'s functions are constructors? How many rules are needed to describe *Library*'s semantics? [5 marks]

ANSWER:

- constructors: *newLib*, *addBook*, *addReader*, *addLoan*
- observers: *getBooks*, *getReaders*, *getLoans*
- mutators: *delBook*, *delReader*, *delLoan*

There are 4 constructors and 6 non-constructors, so there will be 24 (=4 $\times$ 6) rules.

e) Write down the semantics of the required ADT, and explain what each rule means in plain English. [30 marks]

ANSWER:

We need to explain the effect of applying each non-constructor following each constructor. We list the semantics in separate sections, corresponding to the constructor(s) in question.

newLib:

$$\begin{array}{lll} getBooks\ \ newLib & = & noBooks \\ getReaders\ \ newLib & = & noReaders \\ getLoans\ \ newLib & = & noLoans \\ delBook\ \ newLib\ b & = & errNoSuchBook \\ delReader\ \ newLib\ r & = & errNoSuchReader \\ delLoan\ \ newLib\ b\ r & = & errNoSuchLoan \end{array}$$

A new library contains no books, readers or loans. Attempting to delete items of each of these entry types from an empty library generates the associated *no such entry* error.

addBook, **addReader**, **addLoan**:
We give the rules for *addBook*. The rules relating to *addReader* and *addLoan* are analogous, except that *addLoan* should also check whether the specified book and/or reader actually exist. Notice that

$$\textit{addBook } l\ b$$

itself generates the error value *errDuplicateBook* whenever $b \in \textit{getBooks } l$. We will assume that the error is propagated when this situation arises, and only list the cases below where this is not the case. Although we have not included this functionality below, it is also arguably the case that deleting a book or a reader should automatically cause deletion of any loan associated with them.

getBooks	$(\textit{addBook } l\ b)$	$=$	$\{b\} \cup \textit{getBooks } l$
	b is added to the books already in l		
getReaders	$(\textit{addBook } l\ b)$	$=$	$\textit{getReaders } l$
	the list of readers is unaffected		
getLoans	$(\textit{addBook } l\ b)$	$=$	$\textit{getLoans } l$
	the list of loans is unaffected		
delBook	$(\textit{addBook } l\ b)\ b'$	$=$	if $(b = b')$ then l else $\textit{addBook } (\textit{delBook } l\ b')\ b$
	if we're deleting a book we just added, the library reverts to its original state; otherwise we delete the required book from the library before adding the new one		
delReader	$(\textit{addBook } l\ b)\ r'$	$=$	$\textit{delReader } l\ r'$
	the list of readers is unaffected		
delLoan	$(\textit{addBook } l\ b)\ b'\ r'$	$=$	$\textit{delLoan } l\ b'\ r'$
	the list of loans is unaffected		

2. A programmer wants to implement a *deque* (double-ended queue). A deque consists of a sequence of values – you can insert values at either end of the deque, and (provided the deque isn't empty) you can query and remove values at either end. The programmer wants the ability to perform (at least) the following operations.

   - create : Takes no parameters, and returns the empty deque.
   - addFront, addBack : These add an entry to the relevant end of the deque and return the new deque.
   - empty : Tests whether a deque is or is not empty.
   - removeFront, removeBack : These remove an entry from the relevant end of the deque and return the resulting deque.
   - front : Returns the entry currently at the front of the deque.
   - back : Returns the entry currently at the back of the deque.

   a) Design an ADT called **Deque** that satisfies the programmer's requirements; remember to include the relevant sorts, syntax and semantics.

   ANSWER:

Syntax:

- create :: Deque
- addFront :: Value → Deque → Deque
- addBack :: Value → Deque → Deque
- empty :: Deque → Bool
- removeFront :: Deque → (Deque ∪ ErrDeque)
- removeBack :: Deque → (Deque ∪ ErrDeque)
- front :: Deque → (Value ∪ ErrValue)
- back :: Deque → (Value ∪ ErrValue)

Sorts

- Value – the sort of values to be stored in the deque
- ErrDeque – used to record errors when a deque is expected as output. I'll assume that errDeque is defined as part of this sort.
- ErrValue – used to record errors when a value is expected as output. I'll assume that errValue is defined as part of this sort.
- Bool – boolean values. We'll assume true and false are defined as part of this sort.

Semantics I'll assume that the following 3 functions are constructors: create, addFront, addBack. I'll assume that the other 5 functions are non-constructors. This means I need to write down 15 (= 3 × 5) semantic rules — but in fact I'll need more than this, because I'll be defining rules recursively, and I need to think about the base cases. Technically I ought to consider what happens when functions are applied to error values, but I already know how I intend to implement errors, and this won't require any new types to be defined.

$$
\begin{aligned}
\textsf{empty (create)} &= \textsf{true} \\
\textsf{empty (addFront x d)} &= \textsf{false} \\
\textsf{empty (addBack x d)} &= \textsf{false} \\
\\
\textsf{removeFront (create)} &= \textsf{errDeque} \\
\textsf{removeFront (addFront x create)} &= \textsf{create} \\
\textsf{removeFront (addFront x d)} &= \textsf{d} \\
\textsf{removeFront (addBack x create)} &= \textsf{create} \\
\textsf{removeFront (addBack x d)} &= \textsf{addBack x (removeFront d)} \\
\\
\textsf{removeBack (create)} &= \textsf{errDeque} \\
\textsf{removeBack (addFront x create)} &= \textsf{create} \\
\textsf{removeBack (addFront x d)} &= \textsf{addFront x (removeBack d)} \\
\textsf{removeBack (addBack x create)} &= \textsf{create} \\
\textsf{removeBack (addBack x d)} &= \textsf{addBack x (removeFront d)}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{front\ (create)} &= \mathsf{errValue}\\
\mathsf{front\ (addFront\ x\ d)} &= \mathsf{x}\\
\mathsf{front\ (addBack\ x\ create)} &= \mathsf{x}\\
\mathsf{front\ (addBack\ x\ d)} &= \mathsf{front\ (d)}\\
\\
\mathsf{back\ (create)} &= \mathsf{errValue}\\
\mathsf{back\ (addFront\ x\ create)} &= \mathsf{x}\\
\mathsf{back\ (addFront\ x\ d)} &= \mathsf{back\ (d)}\\
\mathsf{back\ (addBack\ x\ d)} &= \mathsf{x}
\end{aligned}
$$

b) Write down a Haskell implementation of your ADT.

ANSWER:

This is essentially the same as above, except that I've chosen to use the error function to handle errors. This means that ErrValue = Value and ErrDeque = Deque, and no new error types need to be defined. I still need to define the error values.

```
data Deque a = Create | AddFront a (Deque a) | AddBack a (Deque a)
         deriving (Show)

errValue = error "No such value"
errDeque = error "No such deque"

empty (Create) = True
empty (AddFront _ _) = False
empty (AddBack _ _) = False

removeFront (Create) = errDeque
removeFront (AddFront _ Create) = Create
removeFront (AddFront _ d) = d
removeFront (AddBack _ Create) = Create
removeFront (AddBack x d) = AddBack x (removeFront d)

removeBack (Create) = errDeque
removeBack (AddFront _ Create) = Create
removeBack (AddFront x d) = AddFront x (removeBack d)
removeBack (AddBack _ Create) = Create
removeBack (AddBack _ d) = d

front (Create) = errValue
front (AddFront x _) = x
front (AddBack x Create) = x
front (AddBack _ d) = front d

back (Create) = errValue
back (AddFront x Create) = x
back (AddFront _ d) = back d
back (AddBack x _) = x
```