Imperative Programming (Parts 1&2): Sheet 4

Joe Pitt-Francis

Most questions by Mike Spivey, adapted by Gavin Lowe

Suitable for around Week 5, Hilary term, 2019

- Any question marked with † is a self-study question and an answer is provided at the end. Unless your tutor says otherwise, you should mark your attempts at these questions, and hand them in with the rest of your answers.
- Questions marked [**Programming**] are specifically designed to be implemented. Unless your tutor says otherwise, provide your tutor with evidence that you have a working implementation.
- You should give an invariant for each non-trivial loop you write, together with appropriate justification.

Question 1

[**Programming**] The Scala library's `HashSet` is a class which extends Scala's own `trait Set` and implements many common set functions[1]. For example, we can easily add an element to an (`Int`) Set and then probe the set for membership.

```scala
val myset = new scala.collection.mutable.HashSet[Int]
myset.add(1);  myset.add(1); myset.add(2)
assert(myset.contains(3)==false)
```

Write a small test suite (with ScalaTest, if possible) which first defines a `val` of type `HashSet[String]`. Then test that the methods `add`, `remove`, `contains`, `size`, and `isEmpty` behave as you expect them to. Can you show any difference in behaviour if you attempt to `add` an element which is already in the set, or if you `remove` an element which is not in the set? Can you write a test (of any function of the `HashSet`) which intercepts an exception thrown by the `HashSet` when a precondition is violated?

[1]See https://www.scala-lang.org/api/2.12.3/scala/collection/mutable/HashSet.html

Question 2

A *stack* is a data structure that stores a sequence of data. A new value `x` may be added to the stack using the operation `push(x)`. The operation `pop` removes the most-recently added value and returns it. The operation `isEmpty` tests whether the stack is empty.

Write an abstract specification of stacks in terms of a trait `Stack[A]`, where `A` is the type of the underlying data. You may use any standard Haskell functions in your specification. For example, it is convenient to use Haskell `++` list concatenation in the specification of a sequence.

Question 3

Consider the following specification of a set of `Int`s.

```
/** state: S : ℙ Int
  * init: S = {} */
trait IntSet{
  /** Add elem to the set.
    * post: S = S₀ ∪ {elem} */
  def add(elem: Int)

  /** Does the set contain elem?
    * post: S = S₀ ∧ returns elem ∈ S */
  def isIn(elem: Int): Boolean

  /** Remove elem from the set.
    * post: S = S₀ − {elem} */
  def remove(elem: Int)

  /** The size of the set.
    * post: S = S₀ ∧ returns #S */
  def size : Int
}
```

(a) Suppose we want to restrict the set to contain elements from $[0..N)$. Sketch the necessary changes to the specification.

(b) One way to implement such a set is to use an array `a` of booleans such that `a(x)` is true precisely if `x` is in the set. This data structure is often called a *bit map*, since each element of the array could be a single bit.

Implement a class `BitMapSet` following this idea. Write down a suitable abstraction function and datatype invariant.

Question 4

The scala `Set[A]` trait contains an operation

```
  def head : A
```

The API documentation describes it as

returns the first element of this set.

(a) Comment on the description of the operation.

(b) Write down a suitable specification for the operation (assuming a state as in Question 3).

(c) Give an implementation for this operation for the `BitMapSet` class of Question 3.

Question 5 †

The interface for the "phone book" object is given in Figure 1.

```
/** The state of a phone book, mapping names (Strings) to numbers (also
  * Strings).
  * state:  book : String ⇸ String
  * init:  book = {} */
trait Book{
  /** Add the maplet name -> number to the mapping.
    * post:  book = book₀ ⊕ {name → number} */
  def store(name: String, number: String)

  /** Return the number stored against name.
    * pre:  name ∈ dom book
    * post:  book = book₀ ∧ returns  book(name) */
  def recall(name: String) : String

  /** Is name in the book?
    * post:  book = book₀ ∧ returns  name ∈ dom book */
  def isInBook(name: String) : Boolean
}
```

Figure 1: The `Book` trait

(a) Augment the trait with an operation

```
  /** Delete the number stored against name (if it exists) */
  def delete(name: String) : Boolean = ...
```

The procedure should return `true` precisely if `name` was previously in the phone book. Give a specification on the abstract state space in terms of pre- and post-conditions.

(b) One version of the phone book developed in the lectures (which can be downloaded as `ArrayBooks.scala`) uses a pair of unordered arrays, `names` and `numbers`. Implement the `delete` operation in this version.

Question 6

Implement a version of the phone book that keeps `names` as an *ordered* array (with a matching array of `numbers`), and uses binary search to locate names quickly. Include the procedure `delete` also. What is the complexity (in $O(\_)$ notation) of each of the operations?

Question 7

[Programming] Suppose we want to represent a bag of integers from the range $[0..\mathtt{MAX})$. Abstractly, this bag can be thought of as having a state

state: $bag : Int \to Int$

such that $bag(x)$ gives the number of occurrences of x in the bag. Write a trait `Bag` specifying a bag in terms of such a state. Include operations to add an integer, and to find the number of copies of an integer in the bag.

Now provide an implementation using an array

```
val c = new Array[Int](MAX);
```

such that `c(i)` records the number of copies of `i` in the bag. Include a suitable abstraction function.

Question 8

Suppose we want to sort a collection of `N` numbers in the range `[0..MAX)`. Show how we can do so in time $\Theta(\mathtt{N} + \mathtt{MAX})$. Hint: use your answer to Question 7. [2]

Answer to question 5 †

(a)
```
/** Delete the number stored against name (if it exists).
  * post: returns (name ∈ dom book₀) ∧
  *       (name ∈ dom book₀ ∧ book = book₀ − {name → book₀(name)} ∨
  *        name ∉ dom book₀ ∧ book = book₀) */
def delete(name: String) : Boolean
```

If the supplied key is not in the phone book, this procedure leaves the state unchanged. The user interface could give a suitable message to the user depending on the value returned. (Also, returning a boolean value here is consistent with many collection classes in the Scala API.)

(b) As with `store` and `recall`, the subroutine `find` comes in useful:

[2] **[Programming]** You may also wish to write and test a program for this question which will further verify your program from Question 7.

```
/** Delete the number stored against name (if it exists) */
def delete(name: String) : Boolean = {
  val i = find(name)
  if(i != count){
    // Plug the gap with the final entry (because ordering does not matter)
    count = count-1;
    names(i) = names(count)
    numbers(i) = numbers(count); true
  }
  else false
}
```