CSci 450: Org. of Programming Languages
CSci 503: Fundamental Concepts in Languages
Assignment #4, Fall 2017

H. Conrad Cunningham

Revised 18 October 2017

Assignment #4 (Optional)
Due Monday, 30 October, 11:59 p.m.

This assignment can be used to replace Assignment #1, #2, or #3.

This assignment period may overlap with that of Assignment #5.

General Instructions

All homework and programming exercises must be prepared in accordance with the instructions given in the Syllabus. Each assignment must be submitted to your instructor by its stated deadline.

Citations: In accordance with expected scholarly and academic standards, if you reference outside textbooks, reference books, articles, websites, etc., or discuss an assignment with individuals inside or outside the class, you must document these by including appropriate citations or comments at prominent places in your submission such as in the header of the primary source file.

Identification: Put your name, course name, and assignment number as comments in each file you submit.

Assignment Description

1. This is an individual assignment. When complete, submit your Haskell source and testing modules to the course Blackboard site.

   Be sure to document your code appropriately using program comments. Give attention to the general instructions given above and in the Syllabus.

2. On exercise 14 in Chapter 4 of *Introduction to Functional Programming Using Haskell* (shown below) do at least 10 of the 16 parts.

Put your functions in a module named `Bag` (in file `Bag.hs`).

Although this exercise is given in Chapter 4, you may use Haskell features and techniques from Chapters 5 (higher-order functions), 7 (list comprehensions), and 8 (algebraic data types) as appropriate. You can use standard Haskell libraries such as `Data.List` (e.g., `sort`, `replicate`).

3. Test your module thoroughly. Put your testing code in a module named `BagTest` (in file `BagTest.hs`).

Chapter 4 Exercises

14. Bag module.

    Mathematically, a *bag* (or *multiset*) is a function from some arbitrary set of elements (the domain) to the set of nonnegative integers (the range). We interpret the nonnegative integer as the number of occurrences of the element in the bag. Zero means the element does not occur.

    From another perspective, a bag is an unordered collection of elements. Each element may occur one or more times in the bag. (It is like a set except values can occur multiple times.)

    For example, `{| "time', "time", "and", "time", "again" |}` is a bag containing 5 strings. There are 3 occurrences of string `"time"` and 1 occurrence each of strings `"and"` and `"again"`.

    `{| 11, 2, 3, 7, 5 |}` is a bag of prime numbers. It is also a set because each element occurs exactly once.

    We can represent a bag in many ways in Haskell. Using lists, we could represent a bag with a simple (unordered) list of elements, an ordered list of elements, an unordered or an ordered list of tuples which pair an element with the (nonzero) number of times it occurs, etc. A bag could also be represented with other data structures such as a `Map` from library `Data.Map`.

    Choose some representation for polymorphic bags. You may assume that the elements in the domain are totally ordered (i.e., are from a type that is an instance of class `Ord`), but otherwise the elements can be of any type.

    For example, if you use a list representation, you might define the type synonym:

    ```
    type Bag a = [a]
    ```

    Develop a data abstraction (information-hiding) module that encapsulates the representation of the data structure used to store the elements inside the module.

    The module should include the following public functions. This interface should be the same even if you change the representation of the data internally.

    a. `newBag` returns a new bag with no elements (i.e., empty).

    b. `listToBag` takes a list of elements and returns a bag containing exactly those elements. The

number of occurrences of an element in the list and in the resulting bag is the same.

c. `bagToList` takes a bag and returns a list containing exactly the elements occurring in the bag. The number of occurrences of an element in the bag and in the resulting list is the same.

   Note: It is not required that `bagToList (listToBag xs) == xs`. But it is required that both sides have the same numbers of the same elements.

d. `isEmpty` takes a bag and returns `True` if the bag has no elements and returns `False` otherwise.

e. `isElem` takes an element and a bag and returns `True` if the element occurs in the bag and returns `False` otherwise.

f. `size` takes a bag and returns its cardinality (i.e., the total number of occurrences of all elements).

g. `occursBag` takes an element and a bag and returns the number of occurrences of the element in the bag.

h. `insertElem` takes an element and a bag and returns the bag with the element inserted. Bag insertion either adds a single occurrence of a new element to the bag or increases the number of occurrences of an existing element by one.

i. `deleteElem` takes an element and a bag and returns the bag with the element deleted. Bag deletion removes a single occurrence of an element from the bag, decreases the number of occurrences of an existing element by one, or does not change the bag if the element does not occur.

j. `eqBag` takes two bags and returns `True` if the two bags are equal (i.e., the same elements and same number of occurrences of each) and returns `False` otherwise.

   Note: If `bagToList xs == bagToList ys`, then `eqBag xs ys`. However, if `eqBag xs ys`, it is not required that `bagToList xs == bagToList ys`.

k. `unionBag` takes two bags and returns their bag union. The union of bags X and Y contains all elements that occur in either X or Y; the number of occurrences of an element in the union is the number in X or in Y, whichever is greater.

l. `intersectBag` takes two bags and returns their bag intersection. The intersection of bags X and Y contains all elements that occur in both X and Y; the number of occurrences of an element in the intersection is the number in X or in Y, whichever is lesser.

m. `sumBag` takes two bags and returns their bag sum. The sum of bags X and Y contains all elements that occur in X or Y; the number of occurrences of an element is the sum of the number of occurrences in X and Y.

n. `diffBag` takes two bags and returns the bag difference, first argument minus the second. The difference of bags X and Y contains all elements of X that occur in Y fewer times; the number of occurrences of an element in the difference is the number of occurrences in X minus the number in Y.

o. `subBag` takes two bags and returns `True` if the first is a subbag of the second and `False` otherwise. X is a subbag of Y if every element of X occurs in Y at least as many times as it does in X.

p. `bagToSet` takes a bag and returns a list containing exactly the *set* of elements contained in the bag. Each element occurring one or more times in the bag will occur exactly once in the list returned.