EECS3311 Lab4 Abstract Data Type BAG[G]

1. Abstract Data Type BAG[G] 1
2. Design Learning Outcomes 2
Electronic submission 4
Void safety 5
Written Report 6
3. Design Decisions 6

1. Abstract Data Type BAG[G]

A chart view of the BAG abstract data type is provided below:

```
deferred class
        ADT_BAG [G -> {HASHABLE, COMPARABLE}]

General
        cluster: bag
        description: "Abstract Data Type for BAG[G] where G is hashable and comparable"

Ancestors
        ITERABLE* [G]

Queries
        bag_equal alias "|=|" (other: [like Current] attached ADT_BAG [G]): BOOLEAN
        count: INTEGER_32
        domain: ARRAY [G]
        has (a_item: G): BOOLEAN
        is_nonnegative (a_array: ARRAY [TUPLE [G, INTEGER_32]]): BOOLEAN
        is_subset_of alias "|<:" (other: [like Current] attached ADT_BAG [G]): BOOLEAN
        new_cursor: ITERATION_CURSOR [G] -- (from ITERABLE)
        number_of (f: PREDICATE [ANY, TUPLE [G, INTEGER_32]]): INTEGER_32
        occurrences alias "[]" (key: G): INTEGER_32
        total: INTEGER_32

Commands
        add_all (other: [like Current] attached ADT_BAG [G])
        extend (a_key: G; a_quantity: INTEGER_32)
        remove (a_key: G; a_quantity: INTEGER_32)
        remove_all (other: [like Current] attached ADT_BAG [G])

Constraints
        consistent count
        nonnegative items
        reflexivity
```

Mathematically, we may think of a *bag* as {["nuts", 2], ["bolts", 5]}, i.e. a mapping from a type G (in this case STRING) to a natural number. So in this bag we have 2 nuts and 5 bolts. The type of bag in generic parameter G is

$\boxed{bag: G \rightarrow \mathbb{N}}$

In the example, we chose BAG[STRING] thus

$\boxed{bag: \text{STRING} \rightarrow \mathbb{N}}$

2. Design Learning Outcomes

In this Lab, you are provided with a deferred bag class ADT_BAG[G]. You must implement the bag ADT, i.e. you must construct a class MY_BAG[G] that inherits from ADT_BAG and implements all the deferred features which will include

- the normal bag operations such as *has*, *domain*, *occurrences*, *extend*, *remove*, *is_subset_of*, etc.
- the ability to form a bag directly from an array of tuples (in which case the array may not have duplicates) using the **convert** notation.
- support the **across** iterator using the iterator design pattern
- support the counting quantifier (*number_of)* using agents, i.e. lambda expressions.

The example below shows some of the notation

```
bag1: MY_BAG[STRING]
bag2: like bag1
b1, b2: BOOLEAN

make
        -- Run some bag stuff
    do

        bag1 := <<["nuts", 2], ["bolts", 5]>>

        bag2 := <<["nuts", 2], ["bolts", 6], ["hammers", 5]>>

        check bag1.has ("nuts") and then bag1["nuts"] = 2 end


        --  (∀p ∈ bag1 : (p = "nuts" ∨ p = "bolts") ∧ bag1[p] ≥ 2)
        b1 := across bag1 as it all
            it.item ~ "nuts" or it.item ~ "bolts"
            and then bag1["nuts"] >= 2
            and then bag1[it.item] >= 2
        end

        -- (∃p ∈ bag1 : p = "nuts")
        b2 := across bag1 as it some
            it.item ~ "nuts"
        end

        check b1 and b2 end

        --bag2 is a subset of bag1:  bag2 ⊆ bag1
        check bag1 |<: bag2 end

        print ("If you got this far, all is ok%N")

    end
```

Below we see more uses of the various notations

```
bag1: MY_BAG[STRING]
bag2: like bag1
b1, b2: BOOLEAN

make
        -- Run some bag stuff
    do

        bag1 := <<["nuts", 2], ["bolts", 5]>>

        bag2 := <<["nuts", 2], ["bolts", 6], ["hammers", 5]>>

        check bag1.has ("nuts") and then bag1["nuts"] = 2 end


        --  (∀p ∈ bag1 : (p = "nuts" ∨ p = "bolts") ∧ bag1[p] ≥ 2)
        b1 := across bag1 as it all
            it.item ~ "nuts" or it.item ~ "bolts"
            and then bag1["nuts"] >= 2
            and then bag1[it.item] >= 2
        end

        -- (∃p ∈ bag1 : p = "nuts")
        b2 := across bag1 as it some
            it.item ~ "nuts"
        end

        check b1 and b2 end

        --bag2 is a subset of bag1:  bag2 ⊆ bag1
        check bag1 |<: bag2 end

        -- (#[g,i] ∈ bag2 : i ≥ 5) = 2
        check bag2.number_of(agent gt5) = 2 end

        --inline agent
        check bag2.number_of (agent (g: STRING; i:INTEGER): BOOLEAN do Result := i >= 5 end) = 2 end

        print ("If you got this far, all is ok%N")

    end

gt5(g:STRING;i:INTEGER): BOOLEAN
        -- Is number of items greater than or equal to 5
    do
        Result := i >= 5
    end
```

across iterator design pattern

bag subset operator

bag counting quantifier

inline agent

Error correction: In the above, it should say bag1 ⊆ bag2.

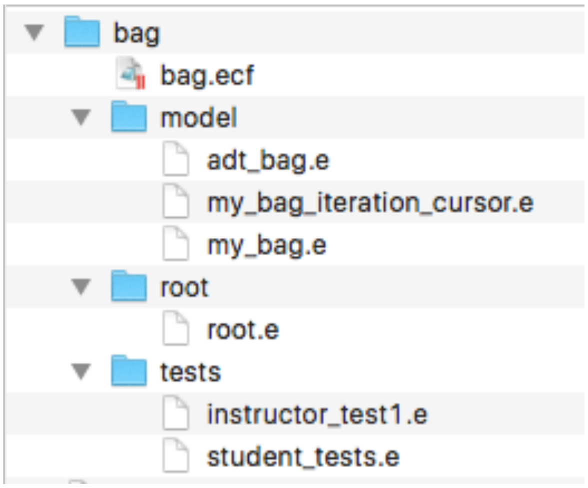
A bag *bag1* is a subset of another bag *bag2* iff for any element *p* that occurs *n* times in *bag1*, the element *p* also occurs at least *n* times in *bag2*.

The *number_of* feature of class BAG is a counting quantifier, constructed by using functional programming (also called the *lambda calculus*). If *f* is a function, then the notation **agent** *f* is an object that stores the function. Suppose you do *a := agent f,* then *a.call*([x,y]) has the same effect as if you directly called *f*(x,y) for any applicable arguments *x* and *y*. The feature *call* is applicable to all agents; it takes a single TUPLE, here [x,y], as argument. In bag2, there are at least five occurrences (i ≥ 5) of bolts and hammer; thus the counting quantifier returns two items.[1]

1. See http://eiffel.eecs.yorku.ca/ for more on agents (and also the newer notation).

In ESpec, we store tests in this way and the call them (execute them) when we are ready to run the test suite. Read more about this in *Touch of Class*, chapter 17 (available online via the Steacie library). Functional programming is particularly useful in event driven design (chapter 18). How to submit

Electronic submission



Submit a directory *bag* that is structured precisely as shown above. You are provided with the abstract data type ADT_BAG[G] and some unit tests INSTRUCTOR_TEST1 to get you started. In the cluster bag, you may add additional classes as needed. You must supply at least **three** tests of your own in class STUDENT_TEST (but you really need many more).

- *eclean*, recompile and check that all the tests run
- *eclean* again
- **submit -l 3311 Lab4 bag**

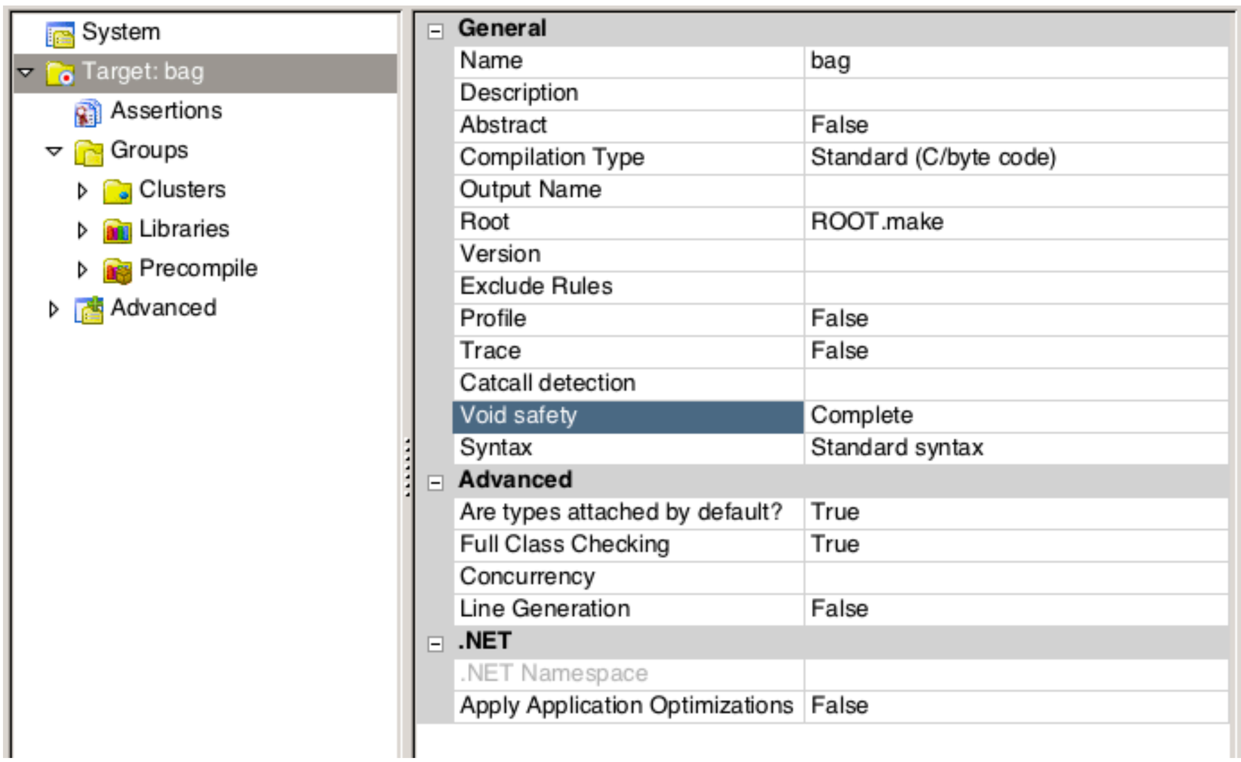
Tests

Tests must be written with the comment in the required style:

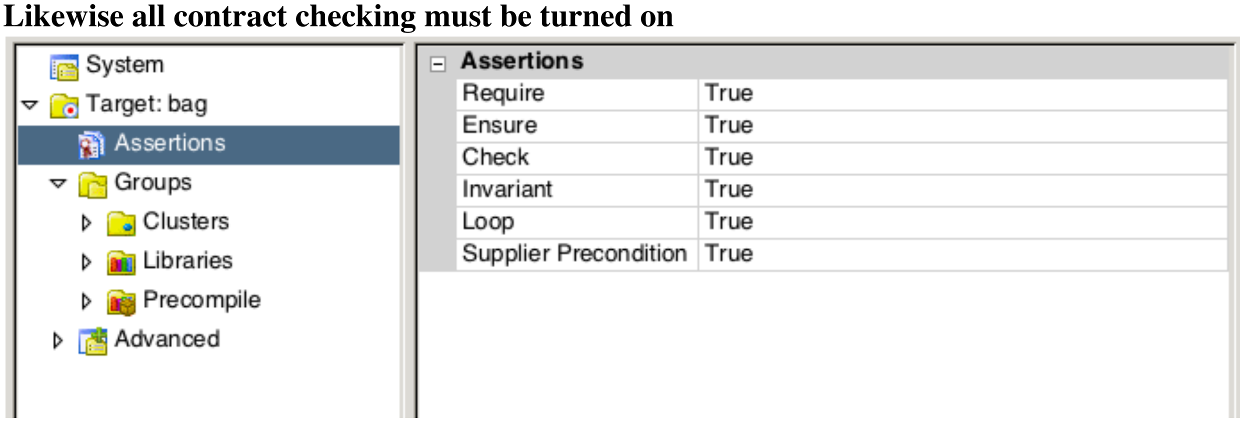
```
t6: BOOLEAN
    local
        bag: MY_BAG [STRING]
    do
        comment ("t6:repeated elements in contruction")
        bag := <<["foo",3], ["bar",3], ["foo",2], ["bar",0]>>
        Result := bag ["foo"] = 5
        check Result end
        Result := bag ["bar"] = 3
        check Result end
        Result := bag ["baz"] = 0
    end
```

Void safety

Ensure that the project settings in the ECF file specify void safety. You can check the ECF from the settings in the IDE, which should be as follows:



Likewise all contract checking must be turned on



Written Report

A written report must also be submitted to Moodle. This report is two pages (no more).

Page 1: Your name, Prism a/c number as usual. Also your statement as to whether you completed the Lab, and if not, why not? Also you must affirm that all the work for this Lab is your own.

Page1 also contains the top-level BON diagram of your design as generated by the EiffelStudio IDE. A clear, clean, screenshot of this IDE diagram must be in your document. Note that by zooming in (via the diagram tool) you can obtain a clean PNG. Your BON diagram must contain: ADT_BAG, MY_BAG, and the iterable, and two cursor classes at the very least.

Page2: Using Visio or Libreoffice templates, provide the same top-level diagram of your design. ADT_BAG must be detailed while the other classes can be compressed. You decide how much information to display and what is critical to display to help another developer understand the design. Don't display too much (that will pollute the diagram) and not too little.

3. Design Decisions

- You may want to consider using a HASH_TABLE for the implementation. This needs to be private. But there are many other implementation possibilities.
- You will need to provide the appropriate iterator machinery. See the iterator design pattern discussed in detail in class.
- Note that the *domain* return type is a *sorted* array.

The **convert** notation is shown at work below.

```
class
    MY_BAG [G -> {HASHABLE, COMPARABLE}]
inherit
    ADT_BAG[G]

    DEBUG_OUTPUT

create
    make_empty,
    make_from_tupled_array

convert
    make_from_tupled_array ({attached ARRAY [attached TUPLE [G, INTEGER_32]]})
```

DEBUG_OUTPUT allows you to provide a string display in the debugger of bags, e.g.

Current object	{[bolts,6],[hammers,5],[nuts,2],}	MY_BAG [!STRING_8]
rep	<0x110A3B068>	HASH_TABLE [INTEGER_32, !STRING_8]
Once routines		
Arguments		
other	{[bolts,6],[hammers,5],[nuts,2],}	MY_BAG [!STRING_8]

The Current object in the debugger represents a bag as a string {[bolts,6][hammers,5]} etc. You do this by implementing *debug_output*: STRING.

Note that we do not use *is_equal* for bag equality but rather we define our own version of bag equality with the alias "|=|". See test 4.

You must write many tests of your own but start by getting the supplied tests working

<table>
<tr><td colspan="3"></td></tr>
<tr><td colspan="3">PASSED (6 out of 6)</td></tr>
<tr><th>Case Type</th><th>Passed</th><th>Total</th></tr>
<tr><td>Violation</td><td>0</td><td>0</td></tr>
<tr><td>Boolean</td><td>6</td><td>6</td></tr>
<tr><td>All Cases</td><td>6</td><td>6</td></tr>
<tr><th>State</th><th>Contract Violation</th><th>Test Name</th></tr>
<tr><td>Test1</td><td colspan="2">INSTRUCTOR_TEST1</td></tr>
<tr><td>PASSED</td><td>NONE</td><td>t1: test bag has, across, subset
bag1 = <<[nuts, 2], [bolts, 5]>>
bag2 = <<[nuts, 2], [bolts, 6], [hammers, 5]>>
check: across bag1 count >= 2
check: across bag2 exists hammers
check ag subset: bag1 |<: bag2 and not bag2 |<: bag1</td></tr>
<tr><td>PASSED</td><td>NONE</td><td>t2: test counting quantifier (#[g,i] in bag2 : i >= 5) = 2</td></tr>
<tr><td>PASSED</td><td>NONE</td><td>t3: test sorted domain</td></tr>
<tr><td>PASSED</td><td>NONE</td><td>t4: extend bag, then check is_equal, count and total</td></tr>
<tr><td>PASSED</td><td>NONE</td><td>t5: test add_all, remove all, remove</td></tr>
<tr><td>PASSED</td><td>NONE</td><td>t6: repeated elements in contruction</td></tr>
</table>