Mock Exam 1

ETH Zurich

November 6, 2013

Name: __________________________________________________________

Group: __________________________________________________________

<table>
<thead>
<tr>
<th>Question</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>Question 1</td>
<td>10</td>
</tr>
<tr>
<td>Question 2</td>
<td>14</td>
</tr>
<tr>
<td>Question 3</td>
<td>16</td>
</tr>
<tr>
<td>Total</td>
<td>40</td>
</tr>
</tbody>
</table>
1 Multiple choice (10 points)

Put checkmarks in the checkboxes corresponding to the correct statements. There is at least one correct answer per question. A correctly checked or unchecked box is worth 0.5 points. An incorrectly checked or unchecked box is worth 0 points. Completely unanswered questions are worth 0 points.

Example:

Which of the following statements are true?

a. The sun is a mass of incandescent gas.  ☒ 0.5 points
b. $2 \times 4 = 8$  ☐ 0 points
c. “Rösti” is a kind of sausage.  ☒ 0 points
c. C is an object-oriented programming language.  ☐ 0.5 points

1. Control structures and recursion.
   a. If we know that a loop decreases its variant and that it never goes below 5, then we know that the loop terminates.  ☐
   b. The loop invariant may be violated during the loop initialization (before entering the loop itself).  ☐
   c. The loop invariant tells us how many times the loop will be executed.  ☐
   d. In Eiffel a procedure can have an empty body (do end).  ☐
   e. A loop can always be rewritten as a finite sequence of conditional statements and compound statements.  ☐

2. Inheritance and polymorphism.
   a. All classes in Eiffel implicitly inherit from class OBJECT.  ☐
   b. At runtime a variable can be attached to an object, whose dynamic type inherits from the variable’s static type.  ☐
   c. At runtime a variable can be attached to an object, whose dynamic type is the same as the variable’s static type.  ☐
   d. At runtime a variable can be attached to an object, whose dynamic type is an ancestor of the variable’s static type.  ☐
   e. For an object $obj$, the feature call $obj. is_equal(obj)$ can return False.  ☐

3. Objects and classes
   a. All entities store references to run-time objects.  ☐
   b. Different entities can reference the same object.  ☐
   c. Clients of a class $X$ can see all features declared in class $X$.  ☐
   d. A class needs to tell its clients whether a query is an attribute or a function.  ☐
   e. Objects can be created from every class.  ☐

4. Design by Contract
   a. The creation procedure only needs to ensure that the invariant of the created object holds at the end of the procedure body.  ☐
   b. Every procedure ensures that the postcondition True holds.  ☐
   c. The class invariant needs to hold before every procedure call.  ☐
   d. A procedure $pp$, that redefines another procedure $p$, needs to ensure the postcondition of procedure $p$.  ☐
   e. A procedure $pp$, that redefines another procedure $p$, can provide a precondition that is stronger than the one given by procedure $p$.  ☐
2 Inheritance and Polymorphism (14 Points)

Classes *SCIENTIST, COMPUTER_SCIENTIST, BIOLOGIST*, and *PET* shown below are part of an application for managing scientists’ social life on the web.

![BON Diagram](image)

Figure 1: BON Diagram

```plaintext
defined class
  SCIENTIST

feature {NONE} -- Initialization
  make (a_name: STRING)
      -- Initialize Current with 'a_name'.
  require
    a_name_exists: a_name /= Void and then not a_name.is_empty
  do
    name := a_name
  ensure
    name_set: name = a_name
  end

feature -- Access
  name: STRING
      -- Current’s name.

feature -- Basic operations
  introduce
      -- Print info about self.
  do
    io.put_new_line
    print ("My name is " + name + "; ")
  end
end

class
  COMPUTER_SCIENTIST
```
inherit
5  SCIENTIST
   redefine
7     introduce
   end
9
create
11  make

feature -- Basic operations
   introduce
   -- Print info about self.
   do
   precursor
   print ("I am a computer scientist.")
   end
end

class
2  BIOLOGIST

inherit
4  SCIENTIST
   rename
6     introduce as express
8   redefine
10     express
end

create
12  make_with_pet

feature {NONE} -- Initialization
16  make_with_pet (a_name: STRING; a_pet: PET)
   -- Initialization for ‘Current’.
18   require
19     name_exists: a_name /= Void and then not a_name.is_empty
20     pet_exists : a_pet /= Void
   do
22     make (a_name)
24     pet := a_pet
   ensure
26     name_set: name = a_name
28     pet_set: pet = a_pet
   end

feature -- Access
30  pet: PET
   -- Current biologist’s pet.

feature -- Basic operations
Example 1:

```lisp
(create {PET}.make ("Bob")).introduce
```
Does the code compile? ☒ Yes     □ No  
Output/error description My name is Bob and I tend to be afraid.

Example 2:

\[ Bob.\text{introduce} \]

Does the code compile? □ Yes     ☒ No  
Output/error description The code does not compile, because "Bob" is an unknown (not declared) identifier.

Task 1

\[
\text{create } a_{\text{scientist}}.\text{make} ("Theo")
\]

\[ a_{\text{scientist}}.\text{introduce} \]

Does the code compile? □ Yes     □ No  
Output/error description

-----------------------------------------------

Task 2

\[
\text{create } a_{\text{computer\_scientist}}.\text{make} ("Heidi")
\]

\[ a_{\text{computer\_scientist}}.\text{introduce} \]

Does the code compile? □ Yes     □ No  
Output/error description

-----------------------------------------------

Task 3

\[
a_{\text{scientist}} := \text{create } \{\text{COMPUTER\_SCIENTIST}\}.\text{make} ("Helen")
\]

\[ a_{\text{scientist}}.\text{introduce} \]

Does the code compile? □ Yes     □ No  
Output/error description

-----------------------------------------------

Task 4

\[
a_{\text{scientist}} := \text{create } \{\text{COMPUTER\_SCIENTIST}\}.\text{make} ("Hal")
\]

\[
a_{\text{computer\_scientist}} := a_{\text{scientist}}
\]

\[ a_{\text{computer\_scientist}}.\text{introduce} \]
Task 5

```java
create a_bioloskist.make_with_pet ("Reto", create {PET}.make ("Toby"))
a_bioloskist . express
```

Does the code compile? □ Yes □ No
Output/error description

Task 6

```java
create a_bioloskist.make_with_pet ("Kandra", create {PET}.make ("Tom"))
a_computer_scientist := a_bioloskist
a_computer_scientist . introduce
```

Does the code compile? □ Yes □ No
Output/error description

Task 7

```java
a_bioloskist := create {BIOLOGIST}.make_with_pet ("Elmo", create {PET}.make ("Hex"))
a_scientist := a_bioloskist
a_scientist . pet . introduce
```

Does the code compile? □ Yes □ No
Output/error description
3 Programming + Contracts (16 points)

In this task you are going to implement several operations for a generic class \( \text{SET} [G] \).

A set is a collection of distinct objects. Every element of a set must be unique; no two members may be identical. All set operations preserve this property. The order in which the elements of a set are listed is irrelevant (unlike for a sequence or tuple). Therefore the two sets \( \{5, 10, 12\} \) and \( \{10, 12, 5\} \) are identical.

There are several fundamental operations for constructing new sets from given sets.

- **Union**: The union of \( A \) and \( B \), denoted by \( A \cup B \), is the set of all elements that are members of either \( A \) or \( B \).

- **Intersection**: The intersection of \( A \) and \( B \), denoted by \( A \cap B \), is the set of all elements that are members of both \( A \) and \( B \).

- **Relative complement of \( B \) in \( A \)** (also called the set-theoretic difference of \( A \) and \( B \)), denoted by \( A \setminus B \) (or \( A - B \)), is the set of all elements that are members of \( A \) but not members of \( B \).

The Jaccard index (or coefficient) measures similarity between sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets (see Figure 2). If both sets are empty the Jaccard coefficient is defined as 1.0.

\[
J(A, B) = \frac{|A \cap B|}{|A \cup B|}
\]

Figure 2: Jaccard index definition for non-empty sets \( A \) and \( B \).

Your task is to fill in the gaps of class \( \text{SET} [G] \) below. Please note:

- Your code should satisfy the contracts and provide new contracts where necessary.

- The set should never contain \textit{Void} elements.

- The number of dotted lines does not indicate the number of missing contract clauses or code instructions.

- The implementation of class \( \text{SET} [G] \) is based on a list. The list uses object comparison, so features like \textit{has} and \textit{prune} use object equality instead of reference equality. You can use the \texttt{across} syntax to iterate over the elements of a \texttt{LIST}. The following features of class \texttt{LIST} may be useful:

```plaintext
class LIST [G] feature
has (v: G): BOOLEAN  -- Does current include 'v'?
extend (v: G)        -- Add 'v' to the end.
prune (v: G)         -- Remove an occurrence of 'v', if any.
-- Other features are omitted.
end
```

---

---
class
  SET [G]
create
  make_empty
feature {NONE} -- Initialization
  make_empty
    -- Create empty Current.
    do
      create {ARRAYED_LIST} content.make (0)
      content.compare_objects
    ensure
      empty_content: content.is_empty
    end
feature -- Access
  count: INTEGER
    -- Cardinality of the current set.
    do
      Result := content.count
    end
  is_empty: BOOLEAN
    -- Is current set empty?
    do
      .................................................................
      .................................................................
      .................................................................
    end
  has (v: G): BOOLEAN
    -- Does current set contain 'v'?
    require
      .................................................................
      .................................................................
    do
      .................................................................
      .................................................................
      .................................................................
    end
end

add (v: G)
    -- Add ‘v’ to the current set.
require

end

remove (v: G)
    -- Remove ‘v’ from the current set.
require

ensure
duplicate: like Current
   -- Deep copy of Current.
do
   create Result.make_empty
   across content as c
   loop
      Result.add (c.item)
   end
ensure
   same_size: Result.count = count
   same_content: across content as c all Result.has (c.item) end
end

feature -- Set operations.

union (another: like Current): like Current
   -- Union product of the current set and ‘another’ set.
require
   ...
do
   ...

intersection (another: like Current): like Current
   -- Intersection product of the current set and ‘another’ set.
require

ensure

end

difference (another: like Current): like Current
   -- Set-theoretic difference of the current set and ‘another’ set.
require

ensure

end
do

ensure

end

feature  --  Set metrics.

jaccard_index (another: like Current): REAL_64
  --  Jaccard similarity coefficient between current set and ‘another’ set.
require

do
```
ensure

end

feature {NONE} -- Implementation
  content: LIST[G]
     -- Items of the set.

invariant
  content_exists: content /= Void
  content_object_comparison: content.object_comparison
  non_negative_cardinality: count >= 0

end
```