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 \textit{OBJECT}. □
   b. At runtime a variable can be attached to an object, whose dynamic type inherits from the variables’s static type. □
   c. At runtime a variable can be attached to an object, whose dynamic type is the same as the variables’s static type. □
   d. At runtime a variable can be attached to an object, whose dynamic type is an ancestor of the variables’s static type. □
   e. For an object \textit{obj}, the feature call \textit{obj.is_equal(obj)} can return \textit{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 \textit{X} can see all features declared in class \textit{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 \textit{True} holds. □
   c. The class invariant needs to hold before every procedure call. □
   d. A procedure \textit{pp}, that redefines another procedure \textit{p}, needs to ensure the postcondition of procedure \textit{p}. □
   e. A procedure \textit{pp}, that redefines another procedure \textit{p}, can provide a precondition that is stronger than the one given by procedure \textit{p}. □
Solution

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 variables’s static type. ☒
   c. At runtime a variable can be attached to an object, whose dynamic type is the same as the variables’s static type. ☒
   d. At runtime a variable can be attached to an object, whose dynamic type is an ancestor of the variables’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 \textit{SCIENTIST}, \textit{COMPUTER\_SCIENTIST}, \textit{BIOLOGIST}, and \textit{PET} shown below are part of an application for managing scientists’ social life on the web.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{BON_Diagram.png}
\caption{BON Diagram}
\end{figure}

\begin{verbatim}
deferred class
2 \textit{SCIENTIST}
4 feature \{NONE\} -- Initialization
6 \hspace{1em} make (a\_name: STRING)
      -- Initialize Current with 'a\_name'.
8 \hspace{1em} require
9 \hspace{2em} a\_name\_exists: a\_name \neq Void and then not a\_name.is\_empty
10 \hspace{1em} do
12 \hspace{2em} name := a\_name
13 \hspace{1em} ensure
14 \hspace{2em} name\_set: name = a\_name
16 feature -- Access
18 \hspace{1em} name: STRING
20 \hspace{1em} -- Current's name.
22 feature -- Basic operations
24 \hspace{1em} introduce
26 \hspace{1em} -- Print info about self.
28 \hspace{2em} do
30 \hspace{3em} io.put\_new\_line
32 \hspace{3em} print ("My name is " + name + "; ")
34 \hspace{2em} end
36 class
3 \textit{COMPUTER\_SCIENTIST}
\end{verbatim}
```plaintext
inherit
5  SCIENTIST
    redefine
    introduce
end
9
create
make

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

class
2  BIOLOGIST

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

12 create
  make_with_pet

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

28 feature -- Access
30 pet: PET
   -- Current biologist’s pet.
32 feature -- Basic operations
```
34   express
       -- Print info about self.
36     do
38       Precursor
39         print ("I am a biologist. ")
40         print ("I have a pet. Its name is " + pet.name + ".")
41     end
42 end

1 class
     PET
3 create
5   make
7 feature {NONE} -- Initialization
9       make (pet.name: STRING)
11      -- Initialization for ‘Current’.
12       require
13           pet.name_exists: pet.name /= Void and then not pet.name.is_empty
14       do
15           name := pet.name
16       ensure
17           pet.name_set: name = pet.name
18       end
19 feature -- Access
21       name: STRING
23           -- Current pet’s name.
25 feature -- Basic operations
27       introduce
29      -- Print info about self.
31     do
32       io.put_new_line
34       print ("My name is " + name + " and I tend to be afraid.")
35     end
37 end

Indicate, for each of the code fragments below, if it compiles by checking the corresponding box. If the code fragment does not compile, explain why this is the case and clearly mark the line that does not compile. If the code fragment compiles, specify the text that is printed to the console when the code fragment is executed.

Given the following variable declarations:

\begin{verbatim}
a_scientist : SCIENTIST
a_computer_scientist: COMPUTER_SCIENTIST
a_biologist : BIOLOGIST
\end{verbatim}

Example 1:

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

Example 2:

```java
Bob.introduce
```

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

Grading Scheme

1 Pt: For stating correctly whether it compiles/doesn’t compile.
1 Pt: For providing the correct output (if it compiles) or the reason why it doesn’t compile.

Task 1

```java
create a_scientist.make ("Theo")
a_scientist.introduce
```

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

Creation instruction applies to target of a deferred type.

Task 2

```java
create a_computer_scientist.make ("Heidi")
a_computer_scientist.introduce
```

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

My name is Heidi; I am a computer scientist.

Task 3

```java
a_scientist := create {COMPUTER_Scientist}.make ("Helen")
a_scientist.introduce
```

7
Does the code compile? □ Yes □ No
Output/error description

My name is Helen; I am a computer scientist.

Task 4

\[
\text{a\_scientist} := \text{create \{COMPUTER\_SCIENTIST\}.make ("Hal")}
\]
\[
\text{a\_computer\_scientist} := \text{a\_scientist}
\]
\[
\text{a\_computer\_scientist}.\text{introduce}
\]

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

Source of assignment is not compatible with target.

Task 5

\[
\text{create a\_biologist.make\_with\_pet ("Reto", create \{PET\}.make ("Toby"))}
\]
\[
\text{a\_biologist}.\text{express}
\]

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

My name is Reto; I am a biologist. I have a pet. Its name is Toby.

Task 6

\[
\text{create a\_biologist.make\_with\_pet ("Kandra", create \{PET\}.make ("Tom"))}
\]
\[
\text{a\_computer\_scientist} := \text{a\_biologist}
\]
\[
\text{a\_computer\_scientist}.\text{introduce}
\]

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

Source of assignment not compatible with target.

Task 7

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

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

Unknown identifier ‘pet’
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 \texttt{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 \texttt{has} and \texttt{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:

```java
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
```plaintext
end

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

do
ensure

end

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

do
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

de
ensure

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

do

ensure

end

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

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

3.1 Solution

class SET [G]

create
	make_empty

feature {NONE} −− Initialization

make_empty
−− Create empty Current.
do
	create content.make (0)
	content.compare_objects
ensure
	empty_content: content.is_empty
feature -- Access

count: INTEGER
   -- Cardinality of the current set.
do
   Result := content.count
end

is_empty: BOOLEAN
   -- Is current set empty?
do
   Result := count = 0
end

has (v: G): BOOLEAN
   -- Does current set contain 'v'?
require
   v /= Void
do
   Result := content.has (v)
end

add (v: G)
   -- Add 'v' to the current set.
require
   v /= Void
do
   if not has (v) then
      content.extend (v)
   end
ensure
   in_set_already: old has (v) implies (count = old count)
   added_to_set: not old has (v) implies (count = old count + 1)
end

remove (v: G)
   -- Remove 'v' from the current set.
require
   v /= Void
do
   content.prune (v)
ensure
   removed_count_change: old has (v) implies (count = old count - 1)
   not_removed_no_count_change: not old has (v) implies (count = old count)
   item_deleted: not has (v)
end

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
  another /= Void
do
  Result := another.duplicate
  across content as c
  loop
    Result.add (c.item)
  end
ensure
  not_smaller: Result.count >= count and Result.count >= another.count
end

intersection (another: like Current): like Current
  -- Intersection product of the current set and ‘another’ set.
require
  another /= Void
do
  create Result.make_empty
  across content as c
  loop
    if another.has (c.item) then
      Result.add (c.item)
    end
  end
ensure
  not_bigger: Result.count <= count and Result.count <= another.count
end

difference (another: like Current): like Current
  -- Set-theoretic difference of the current set and ‘another’ set.
require
  another /= Void
do
  create Result.make_empty
  across content as c
  loop
    if not another.has (c.item) then
      Result.add (c.item)
  end

end

ensure
not_bigger_than: Result.count <= count
not_smaller_than: Result.count >= count - another.count
end

feature -- Set metrics.

jaccard_index (another: like Current): REAL_64
-- Jaccard similarity coefficient between current set and 'another' set.
require
another /= Void
do
if not (is_empty and another.is_empty) then
   Result := intersection (another).count / union (another).count
else
   Result := 1.0
end
ensure
bounds: Result >= 0.0 and Result <= 1.0
empty_case: (is_empty and another.is_empty) implies Result = 1.0
end

feature {NONE} -- Implementation

content: ARRAYED_LIST[G]
-- Items of the set.

invariant

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

end