Mock Exam 1

ETH Zurich

November 5, 2014

Name: ________________________________

Group: ________________________________

<table>
<thead>
<tr>
<th>Question</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Question 1</td>
<td>7.5</td>
</tr>
<tr>
<td>Question 2</td>
<td>14</td>
</tr>
<tr>
<td>Question 3</td>
<td>15</td>
</tr>
<tr>
<td>Total</td>
<td>36.5</td>
</tr>
</tbody>
</table>
1 Multiple choice (7.5 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?

- The sun is a mass of incandescent gas. ☒ 0.5 points
- $2 \times 4 = 8$ □ 0 points
- “Rösti” is a kind of sausage. ☒ 0 points
- 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 is checked at the end of 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. 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. □

3. 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 $\textbf{True}$ holds. □
   - c. The class invariant needs to hold before every procedure call. □
   - d. For functions, the precondition may not refer to the result expression and the postcondition may not refer to the arguments of the function. □
   - e. A feature with precondition $\textbf{false}$ is accepted by the compiler. □

1.1 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 is checked at the end of 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. 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. □

3. 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. (!! The statement is ambiguous. Everyone gets 0.5 point !!)
   b. Every procedure ensures that the postcondition $\text{True}$ holds. ☒
   c. The class invariant needs to hold before every procedure call. □
   d. For functions, the precondition may not refer to the $\text{result}$ expression and the postcondition may not refer to the arguments of the function. ☒
   e. A feature with precondition $\text{false}$ is accepted by the compiler. ☒
2 Specifying Software through Contracts (14 points)

A range of integers can be conveniently represented using the boundary values of the range, e.g., the range of integers between \( m \) and \( n \) (inclusive) can be represented using \([m, n]\). Given a range \( R \), we use \( S_R \) to denote the set of integers within \( R \), i.e.

\[
S_{[m,n]} = \{ x \mid m \leq x \leq n \}.
\]

For example, \( S_{[1,3]} = \{1, 2, 3\} \) and \( S_{[3,1]} = \emptyset \).

Listing 1 shows a class \( RANGE \), which abstracts integer ranges and provides functions that operate on them. The preconditions of the functions are already defined in the class; the function results, however, are only given in the comments in terms of the boundary values and the integer sets corresponding to the operand ranges. For example, the comment of function \( is\_equal \) stipulates that \( Result \) should be \( True \) if and only if \( Current \) and \( other \) represent the same set of integers, and the comment of function \( add \) specifies the integer set of \( Result \) should be equal to the union of the sets of \( Current \) and \( other \).

Read through the code, then complete the postconditions so that they reflect the function comments.

Please note:

- The number of dotted lines is not indicative of the number of missing contract clauses.

- You need to write \( True \) at places where you think no explicit contract is necessary: leaving a postcondition empty gives you 0 point for that section.

- The following features from class \( INTEGER \) may be useful:

```java
class INTEGER
    feature
        max (other: INTEGER): INTEGER
            -- The greater of current object and 'other'.
        min (other: INTEGER): INTEGER
            -- The smaller of current object and 'other'.
    end

Listing 1: Class RANGE
```

```
create
    ANY
        redefine is_equal end
```
feature \{NONE\} -- Initialization

\begin{verbatim}
make (l, r : INTEGER)
do
  left := l
  right := r
end
\end{verbatim}

feature -- Access.

\begin{verbatim}
left : INTEGER
  -- Lower boundary of the range.
  -- \( S_{Current} = \{ x \mid left \leq x \leq right \} \)
\end{verbatim}

\begin{verbatim}
right : INTEGER
  -- Upper boundary of the range.
  -- \( S_{Current} = \{ x \mid left \leq x \leq right \} \)
\end{verbatim}

feature -- Query

\begin{verbatim}
is_equal (other : like Current) : BOOLEAN
  -- Result = \( (S_{Current} = S_{other}) \)
require
  other /= Void
ensure
\end{verbatim}

\begin{verbatim}
is_empty : BOOLEAN
  -- Result = \( (S_{Current} = \emptyset) \)
require
  True
ensure
\end{verbatim}

\begin{verbatim}
is_sub_range_of (other : like Current) : BOOLEAN
  -- Result = \( (S_{Current} \subseteq S_{other}) \)
require
  other /= Void
ensure
\end{verbatim}

\begin{verbatim}
is_super_range_of (other : like Current) : BOOLEAN
  -- Result = \( (S_{Current} \supseteq S_{other}) \)
\end{verbatim}
require
  other /= Void

ensure

left_overlaps (other: like Current): BOOLEAN
  --- Result = (left ∈ (S_{Current} ∩ S_{other}))
require
  other /= Void
ensure

right_overlaps (other: like Current): BOOLEAN
  --- Result = (right ∈ (S_{Current} ∩ S_{other}))
require
  other /= Void
ensure

overlaps (other: like Current): BOOLEAN
  --- Result = (S_{Current} ∩ S_{other} ≠ ∅)
require
  other /= Void
ensure

feature -- Operation

\texttt{add} (other: like Current): RANGE
  --- S_{Result} = (S_{Current} \cup S_{other})
require
  other /= Void
  result\_is\_range: is\_empty or other.is\_empty or overlaps (other)
ensure
  Result /= Void
\[ \text{subtract} \ (\text{other}: \text{like Current}): \text{RANGE} \]
\[ S_{\text{Result}} = (S_{\text{Current}} - S_{\text{other}}) \]

**require:**

\[ \text{other} \neq \text{Void} \]
\[ \text{result is range: not overlaps (other)} \]
\[ \text{or left overlaps (other)} \text{ or right overlaps (other)} \]

**ensure:**

\[ \text{Result} \neq \text{Void} \]
2.1 Solution

Listing 2: Class RANGE

```
note
description: "A range of integers."

class RANGE

create make

feature {NONE} -- Initialization
    make (l, r: INTEGER)
    do
       left := l
       right := r
    end

feature -- Access.

left: INTEGER
    -- Lower boundary of the range.
    -- $S_{Current} = \{x \mid left \leq x \leq right\}$

right: INTEGER
    -- Upper boundary of the range.
    -- $S_{Current} = \{x \mid left \leq x \leq right\}$

feature -- Query

is_equal (other: like Current): BOOLEAN
    -- Result = ($S_{Current} = S_{other}$)
    require
       other /= Void
    ensure
       Result = ($is_{empty}$ and other.$is_{empty}$) or
       (left = other.left and right = other.right))

is_empty: BOOLEAN
    -- Result = ($S_{Current} = \emptyset$)
    require
       True
    ensure
       Result = left > right

is_sub_range_of (other: like Current): BOOLEAN
    -- Result = ($S_{Current} \subseteq S_{other}$)
    require
       other /= Void
    ensure
       Result = ($is_{empty}$ or (other.left <= left and right <= other.right))
```
is_super_range_of (other: like Current): BOOLEAN
   \[\text{Result} = (S_{\text{Current}} \supseteq S_{\text{other}})\]
   require
   \[\text{other} /= \text{Void}\]
   ensure
   \[\text{Result} = (\text{other}.\text{is_empty or (left} \leq \text{other}.\text{left and other}.\text{right} \leq \text{right})]\n
left_overlaps (other: like Current): BOOLEAN
   \[\text{Result} = (\text{left} \in (S_{\text{Current}} \cap S_{\text{other}}))\]
   require
   \[\text{other} /= \text{Void}\]
   ensure
   \[\text{Result} = (\text{not is_empty and other}.\text{left} \leq \text{left and left} \leq \text{other}.\text{right})]\n
right_overlaps (other: like Current): BOOLEAN
   \[\text{Result} = (\text{right} \in (S_{\text{Current}} \cap S_{\text{other}}))\]
   require
   \[\text{other} /= \text{Void}\]
   ensure
   \[\text{Result} = (\text{not is_empty and other}.\text{left} \leq \text{right and right} \leq \text{other}.\text{right})]\n
overlaps (other: like Current): BOOLEAN
   \[\text{Result} = (S_{\text{Current}} \cap S_{\text{other}} \neq \emptyset)\]
   require
   \[\text{other} /= \text{Void}\]
   ensure
   \[\text{Result} = \text{not is_empty and not other}.\text{is_empty and}
               (\text{is_sub_range_of (other) or is_super_range_of (other) or}
                \text{left_overlaps (other) or right_overlaps (other)})\]

feature -- Operation

add (other: like Current): RANGE
   \[\text{Result} = (S_{\text{Current}} \cup S_{\text{other}})\]
   require
   \[\text{other} /= \text{Void}\]
   ensure
   \[\text{Result} /= \text{Void}
   \text{is_empty implies Result}.\text{is_equal (other)}
   \text{other}.\text{is_empty implies Result}.\text{is_equal (Current)}
   \text{not (is_empty or other}.\text{is_empty) implies}
   \qquad (\text{Result}.\text{left} = \text{left}.\min (\text{other}.\text{left}) \text{ and}
   \text{Result}.\text{right} = \text{right}.\max (\text{other}.\text{right}))\n
subtract (other: like Current): RANGE
   \[\text{Result} = (S_{\text{Current}} \setminus S_{\text{other}})\]
   require
   \[\text{other} /= \text{Void}\]
   ensure
   \[\text{Result} /= \text{Void}
   \text{is_range : not overlaps (other)
   or left_overlaps (other) or right_overlaps (other)}\]
ensure
  Result /= Void
  not overlaps (other) implies Result.is_equal (Current)
  left_overlaps (other) and not right_overlaps (other) implies
    Result.left = other.right + 1 and Result.right = right
  right_overlaps (other) and not left_overlaps (other) implies
    Result.left = left and Result.right = other.left − 1
  left_overlaps (other) and right_overlaps (other) implies
    Result.is_empty
end
3 Data Structures (15 points)

3.1 Background information

A skip list is a data structure that expands on the idea of a linked list. A node in a linked-list has 1 link; each node in a skip list has 4 links, up, down, left, and right.

A skip list has the following properties:

- The nodes are arranged into rows; each row is a list of sorted elements.
- Every row, except for the bottom row, contains a subset of the elements beneath it, as in Figure 1. This implies that the bottom row contains all the elements in the skip list.
- All nodes are mutually linked, i.e. node_a.left = node_b iff node_b.right = node_a, and likewise for up and down.
- Every row begins with a universal minimal element (represented here by $-\infty$).
- If an element exists in two adjacent rows, then the nodes are linked through the up/down attributes. This can be seen for the elements 20 and $-\infty$ in Figure 1.

![Figure 1: Initial skip list](image1)

When a new element is inserted into the skip list, it is first inserted into the bottom row, as in Figure 2.

![Figure 2: Skip list after insertion of 30 into the bottom row](image2)

Whenever a node is added to any row, there is a chance that it will be promoted, adding it to the row above, as in Figure 3. If there is no row above, a new one will be created. This promotion to the row above happens randomly, and a promotion can trigger another promotion (again, randomly).

3.2 Task

For the task the search feature is already implemented, and returns the rightmost node in the bottom row of the skip list less-than or equal to the argument elem. Feature is_promoted randomly returns True or False, indicating whether to promote a node at any given time. You must implement:
• *insert_in_row* (*a_pre*, *a_node*: `SKIP_LINKABLE`) inserts *a_node* directly after *a_pre* with no promotion. An instance of this can be seen in the transformation between Figure 1 and Figure 2.

• *promote* (*a_link*: `SKIP_LINKABLE`) takes *a_link*, which is already inserted in a row, and either promotes it or does nothing. Remember, *promote* can trigger another promotion.

• *insert* (*elem*: `INTEGER`) takes an element and inserts a new node into the correct position in the skip list, including promotion (if any).

While writing these procedures you are encouraged to use any applicable features already available in the `SKIP_LIST` and `SKIP_LINKABLE` classes (i.e. the features shown below without dotted lines).

```plaintext
class SKIP_LINKABLE
create
make

feature {NONE}
make (*a_value*: `INTEGER`)
    --- Create a node with value ‘a_value’.
    do
      value := a_value
    end

feature --- Set links
set_up (*a_up*: `SKIP_LINKABLE`)
    do
      up := a_up
    end

set_down (*a_down*: `SKIP_LINKABLE`)
    do
      down := a_down
    end

set_left (*a_left*: `SKIP_LINKABLE`)
    do
      left := a_left
    end
```

Figure 3: Skip list after promotion of 30-node
set_right (a_right: SKIP_LINKABLE)
  do
    right := a_right
  end

feature -- Queries
value: INTEGER
  up, down, left, right: SKIP_LINKABLE

invariant
  non_circ: left /= Current and right /= Current
end

class SKIP_LIST

feature
  minimum: INTEGER
    -- Universal minimal element.

  has (elem: INTEGER): BOOLEAN
    -- Does list contain 'elem'?
  do
    -- Implementation omitted.
  end

search (elem: INTEGER): SKIP_LINKABLE
  -- Rightmost node of the bottom row with value <= 'elem'.
ensure
  result_exists: Result /= Void
  result_precedes_element: Result.value <= elem
end

insert (elem: INTEGER)
  -- Insert new node with value 'elem' into the list.
require
  not has (elem)
local
  .................................................................

 .................................................................

do
  .................................................................
ensure
  has (elem)
end

insert_in_row (a_pre, a_node: SKIP_LINKABLE)
  \begin{itemize}
  \item Insert node 'a_node' after node 'a_pre'.
  \end{itemize}
require
  nodes_exist: attached a_pre and attached a_node
different_nodes: a_pre $\neq$ a_node
local
end

promote (a_link: SKIP_LINKABLE)
    -- Possibly promote 'a_link'.
require
    node_exists: attached a_link
    already_inserted: attached a_link.left
local

do
end
end
3.3 Solution

class 
\textit{SKIP\_LIST} [\textit{G} \rightarrow \textit{COMPARABLE}]

\begin{verbatim}
inherit ANY
  redefine out
end

create
make

feature
make (a_signal: G)
do
  create rand.set_seed (42)
  minimum := a_signal
  create head.make (minimum)
end

out: STRING
local
curs: SKIP\_LINKABLE [G]
do
  Result := ""
  from curs := head
  until curs.down = Void
  loop
    curs := curs.down
  end
until curs = Void
loop
  Result := Result + curs.value.out + ","
  curs := curs.right
end
end

minimum: G

head: SKIP\_LINKABLE [G]

has (elem: G): BOOLEAN
  Result := search (elem).value = elem
end

search (elem: G): SKIP\_LINKABLE [G]
\end{verbatim}
local
curs: SKIP_LINKABLE [G]
done: BOOLEAN
do
from curs := head until curs = Void or done
loop
if elem = curs.value then
  from
  until curs.down = Void
  loop curs := curs.down
end

Result := curs
done := True
elseif elem > curs.value then
  if curs.right = Void or else elem < curs.right.value then
    if curs.down = Void then
      Result := curs
      done := True
    end
    curs := curs.down
  else
    curs := curs.right
  end
end

end

ensure
result_exists : Result /= Void
result_precedes_element : Result.value <= elem
end

insert (elem: G)
require
  not has (elem)
local
new_link: SKIP_LINKABLE [G]
do
  create new_link.make (elem)
insert_in_row (search (elem), new_link)
promote (new_link)
ensure
  has (elem)
end

insert_in_row (a_pre, a_node: SKIP_LINKABLE [G])
require
  nodes_exist: attached a_pre and attached a_node
different_nodes : a_pre /= a_node
do
a_node.set_right (a_pre.right)
a_node.set_left (a_pre)

if a_node.right /= Void then
    a_node.right.set_left (a_node)
end

a_pre.set_right (a_node)
end

promote (a_link: SKIP_LINKABLE [G])
require
    node_exists: attached a_link
already_inserted: attached a_link.left
local
curs: SKIP_LINKABLE [G]
new_link: SKIP_LINKABLE [G]
do
    if is_promoted then
        from curs := a_link
        invariant curs /= Void
        until curs.up /= Void or curs.left = Void
        loop curs := curs.left
        end
        curs := curs.up

        create new_link.make (a_link.value)
        insert_in_row (curs, new_link)
        a_link.set_up (new_link)
        new_link.set_down (a_link)
        promote (new_link)
    end
end

rand: RANDOM
do
    Result := (rand.item \ 2) = 0
    rand.forth
end
end