Mock Exam 2

ETH Zurich

December 4, 2013

Name: 

Group: 

<table>
<thead>
<tr>
<th>Question</th>
<th>Score</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. XX 0.5 points
- b. $2 \times 4 = 8$ □ 0 points
- c. “Rösti” is a kind of sausage. X□ 0 points
- c. C is an object-oriented programming language. □ 0.5 points

Solution

1. Data structures.
   - a. Hashtables map keys to values. XX
   - b. Arrays provide constant-time ($O(1)$) access in the worst case. XX
   - c. Hashtables are commonly implemented using binary search trees. □
   - d. Every node in a linked list stores a reference to the next node, if it exists. XX
   - e. Binary trees provide $O(\log n)$ time access in the worst case. □

2. Inheritance and polymorphism.
   - a. In Eiffel, some classes do not share a common ancestor. □
   - b. If class $B$ inherits from class $A$, all of $A$’s features are available to it. XX
   - c. It is impossible to inherit from two classes directly. □
   - d. Depending on the dynamic type of $x$, two calls to $x.f$ may execute different instructions. XX
   - e. If class $B$ inherits from class $A$, then type $A$ conforms to type $B$. □

3. Objects and classes
   - a. All types are either reference or expanded. XX
   - b. If an object is of an expanded type, its fields cannot be modified at runtime. □
   - c. Suppliers of class $C$ can use all the features of class $C$. □
   - d. A class can be both a supplier and a client. XX
   - e. If $C$ is a deferred class, then no entity can exist in a program with static type $C$. □

4. Design by Contract
   - a. An empty postcondition is equivalent to the postcondition True. XX
   - b. An empty precondition is equivalent to the precondition False. □
   - c. When reasoning about a creation procedure $\text{make}$, you are allowed to assume that the class invariant of the object being created holds at the beginning of $\text{make}$. □
   - d. The invariant of a descendant class implies the invariant of its ancestor. XX
   - e. A (non-creation) procedure with an empty contract and an empty body is correct. XX
2 Quadratic Contracts (14 points)

As you probably remember from the school math course, a quadratic equation is an equation of the form

\[ ax^2 + bx + c = 0, \]

where \( x \) is a variable, \( a, b, c \in \mathbb{R} \) are the coefficients, with \( a \neq 0 \).

The standard way of solving a quadratic equation is to first calculate its discriminant \( \Delta \). If \( \Delta > 0 \) the equation has two real solutions, if \( \Delta = 0 \) — a single real solution and if \( \Delta < 0 \) — no real solutions.

2.1 Your Task

Below you will find a skeleton of a class that stores and solves quadratic equations (uninteresting routine bodies are omitted). The class also contains mathematical functions that are useful in the specification and/or implementation of the main features. Your task is to fill in the contracts (preconditions, postconditions and class invariants) according to the description given above and the header comments of the features. Note that the number of dotted lines does not indicate the number of contract clauses you have to provide.

You can use the following operations on real numbers: +, −, *, /, >, ≥, <, ≤. Do not use precise equality (=), as it produces unexpected results on machine floating point numbers. Instead use the function \textit{approx} \((x, y: \text{REAL}): \text{BOOLEAN}\) defined below, which determines whether two real numbers are equal with finite precision \(\varepsilon\) (in other words \(|x - y| < \varepsilon\)).

```java
class QUADRATIC_EQUATION

create

make

feature {NONE} -- Initialization
make (coef_a, coef_b, coef_c: REAL)
  -- Create an equation with coefficients \(\text{coef}_a\), \(\text{coef}_b\), and \(\text{coef}_c\).
  -- Do not solve the equation yet.
  require
    coef_a_nonzero: \text{not} \text{approx} (coef_a, 0.0)
do...
  ensure
    a_set: \text{approx} (a, coef_a)
    b_set: \text{approx} (b, coef_b)
    c_set: \text{approx} (c, coef_c)
    no_solutions_yet: solution_count = 0
end

feature -- Coefficients
  a, b, c: REAL
  -- Quadratic, linear and constant coefficients.

feature -- Math
  abs (x: REAL): REAL
  -- Absolute value of \(x\).
do
```

... 

ensure
correct_result_positive : x >= 0.0 implies approx (Result, x)
correct_result_negative : x < 0.0 implies approx (Result, -x)
end

approx (x, y: REAL): BOOLEAN

-- Is 'x' equal to 'y' with precision 'epsilon'?
do...
ensure
correct_result : Result = (abs (x - y) < epsilon)
end

epsilon: REAL = 1.e-10

-- Precision with which reals are compared.

sqrt (x: REAL): REAL

-- Square root of 'x'.
require
x_non_negative: x >= 0.0
do...
ensure
correct_square : approx (Result * Result, x)
end

feature -- Solutions
solution_count: INTEGER

-- Number of solutions.
solution (i: INTEGER): REAL

-- Solution number 'i'.
require
i_not_too_small : i >= 1
i_not_too_large : i <= solution_count
do
if i = 1 then
   Result := x_1
else
   Result := x_2
end
ensure
is_solution : approx (a * Result * Result + b * Result + c, 0.0)
end

feature -- Basic operations
solve
-- Solve the equation and store correct number of solutions in 'solution_count'.
local
d: REAL
do
\( d := \text{delta} \)

\[
\begin{align*}
  &\text{if } \text{approx} (d, 0) \text{ then} \\
  &\quad \text{solution\_count} := 1 \\
  &\quad x_1 := -b / (2 \ast a) \\
  \text{elseif } d > 0 \text{ then} \\
  &\quad \text{solution\_count} := 2 \\
  &\quad x_1 := (-b + \sqrt{d}) / (2 \ast a) \\
  &\quad x_2 := (-b - \sqrt{d}) / (2 \ast a) \\
  \end{align*}
\]

\text{ensure}

\[
\begin{align*}
  &\text{not approx (delta, 0.0) and delta < 0.0 implies solution\_count = 0} \\
  &\text{approx (delta, 0.0) implies solution\_count = 1} \\
  &\text{not approx (delta, 0.0) and delta > 0.0 implies solution\_count = 2} \\
  \end{align*}
\]

\( \text{delta: REAL} \)

\text{do}

\[
\ldots
\]

\text{end}

\text{feature \{NONE\} -- Implementation}

\( x_1, x_2: \text{REAL} \)

\text{-- Solutions.}

\text{invariant}

\[
\begin{align*}
  &\text{a\_nonzero: not approx (a, 0.0)} \\
  \end{align*}
\]

\text{end}
3 Recursion: Deleting directories (16 Points)

In this question you will work with the FILE class, which represents both directories and regular files. You can iterate through the files contained in a directory using an internal cursor:

```python
from
    directory . start
until
    directory . after
loop
    −− Do something with ‘directory.item’
    directory . forth
end
```

The delete command of class FILE physically deletes the file from disk and changes the value of the exists query on the corresponding FILE object to False. For a directory this command only works if the directory is physically empty (i.e. no files physically exist in the directory).

3.1 Task 1

Take a look at the following procedure delete_all. It deletes a given directory with all its content using recursion:

```python
delete_all ( directory: FILE)
require
directory /= Void and then (directory.exists and directory.is_directory)
do
    from
directory . start
until
    directory . after
loop
    if directory . item . is_directory then
        delete_all ( directory . item)
    else −− regular file
        directory . item . delete
    end
    directory . forth
end
ensure
    not directory . exists
end
```

Your task is to rewrite delete_all so that it does not use recursion (the procedure is not allowed to call itself). You are not allowed to add new features. You are only allowed to call those features of class FILE that are already used in the recursive implementation of delete_all.

You can use the class LIST for this task. An excerpt is given at the end of the question.

Solution

Version 1

```python
delete_all ( directory: FILE)
```
require
directory /= Void and then (directory.exists and directory.is_directory)
local
directories : LIST [FILE]
cur_directory : FILE
do
   -- delete all files
   from
      create directories
directories.extend_back (directory)
directories.start
   until
      directories.after
   loop
      cur_directory := directories.item
      from
      cur_directory.start
      until
      cur_directory.after
   loop
      if cur_directory.item.is_directory then
         directories.extend_back (cur_directory.item)
      else -- normal file
         cur_directory.item.delete
      end
      cur_directory.forth
   end
directories.forth
end
-- delete all directories
from
directories.finish
until
directories.before
loop
directories.item.delete
directories.back
end
ensure
not directory.exists
end

Version 2

delete_all (directory: FILE)
require
directory /= Void and then (directory.exists and directory.is_directory)
local
directories : LIST [FILE]
cur_directory : FILE
do
   from
create directories
    directories . extend_back ( directory )
until
    directories . is_empty
loop
    cur_directory := directories . last
    directories . remove_back
from
    cur_directory . start
until
    cur_directory . after
loop
    if cur_directory . item . is_directory then
        -- Save the current directory and restart the loop
        -- with the subdirectory as ‘cur_directory’
        directories . extend_back ( cur_directory )
        cur_directory := cur_directory . item
        cur_directory . start
    else -- normal file
        cur_directory . item . delete
        cur_directory . forth
    end
end

cur_directory . delete
end
ensure
    not directory . exists
end

3.2 Task 2

With the following example directory and the invocation

delete_all ( create { FILE } . make ( ”C:\Temp\to_del” ) )

please give the order in which the files will be deleted for (a) the given recursive algorithm and
(b) your non-recursive algorithm (e.g.: 3, 6, 7, 8, 9, 2, 5, 4, 1).

Solution

a) 3, 2, 6, 5, 7, 4, 8, 9, 1
b) 8, 9, 3, 7, 6, 5, 4, 2, 1
3.3 LIST [G] (Excerpt)

class LIST [G]

feature -- Access
    first : like item
      -- Item at first position

    item: G
      -- Current item

    last: like item
      -- Item at last position

feature -- Status report
    after: BOOLEAN
      -- Is there no valid cursor position to the right of cursor?

    before: BOOLEAN
      -- Is there no valid cursor position to the left of cursor?

    is_empty: BOOLEAN
      -- Is the list empty?

feature -- Cursor movement
    back
      -- Move to previous item.

    finish
      -- Move cursor to last position. (Go before if empty.)

    forth
      -- Move cursor to next position.

    start
      -- Move cursor to first position. (Go after if empty.)

feature -- Element change
    extend_back (v: like item)
      -- Add 'v' to end. Do not move cursor.

    extend_front (v: like item)
      -- Add 'v' to beginning. Do not move cursor.

    remove_back
      -- Remove last item. Move cursor after if on last.

    remove_front
      -- Remove first item. Move cursor before if on first.

end -- class LIST