Assignment 8: Recursion

ETH Zurich

Due: 21. November 2012

Goals

• Test your understanding of recursion.

• Implement recursive algorithms.

1 An infectious task

You are the boss of a company concerned about health of your employees (especially in winter - the time of flu epidemics). To take a better decision about the company’s health policy, you decide to simulate the spreading of the flu in a program. For this you assume the following model: if a person has a flu, he spreads the infection to only one coworker, who then spreads it to another coworker, and so on.

The following class PERSON models coworkers. The class APPLICATION creates PERSON objects and sets up the coworker structure.

Listing 1: Class PERSON

```plaintext
class PERSON
create
make

feature -- Initialization
make (a_name: STRING)
    -- Create a person named 'a_name'.
```
require
   a_name_valid: 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

   coworker: PERSON

   has_flu: BOOLEAN

feature -- Element change
   set_coworker (p: PERSON)
      -- Set 'coworker' to 'p'.
      require
         p_exists: p /= Void
         p_different: p /= Current
      do
         coworker := p
      ensure
         coworker_set: coworker = p
      end

   set_flu
      -- Set 'has_flu' to True.
      do
         has_flu := True
      ensure
         has_flu: has_flu
      end

invariant
   name_valid: name /= Void and then not name.is_empty
end

Listing 2: Class APPLICATION

class
   APPLICATION

create
   make

feature -- Initialization
   make
      -- Simulate flu epidemic.
      local
         joe, mary, tim, sarah, bill, cara, adam: PERSON
      do
create joe.make ("Joe")
create mary.make ("Mary")
create tim.make ("Tim")
create sarah.make ("Sarah")
create bill.make ("Bill")
create cara.make ("Cara")
create adam.make ("Adam")

joe.set_coworker (sarah)
adam.set_coworker (joe)
tim.set_coworker (sarah)
sarah.set_coworker (cara)

bill.set_coworker (tim)
cara.set_coworker (mary)
mary.set_coworker (bill)
infect (bill)
end

Table 1 shows four different implementations of feature infect, which is supposed to infect a person p and all people reachable from p through the coworker relation.

To do

1. For each version of infect answer the following questions:
   - Does it do what it is supposed to do?
   - If yes, how? (One to two sentences.)
   - If no, why? (One to two sentences.)

   Note: this is a pen-and-paper task; you are not supposed to use EiffelStudio.

2. The class PERSON above assumes that each employee can only infect one coworker. This is unfortunately too optimistic. Rewrite the class PERSON in such a way that an employee can have (and infect) an arbitrary number of coworkers. Implement a correct recursive feature infect for this new setting. Note: you may use a loop to iterate through the list of coworkers.

3. Optional. The coworker structure with at most one coworker forms a (possibly circular) linked list. Which data structure is formed by a coworker structure with multiple coworkers? What kind of traversal do you apply to traverse this structure in the feature infect?

To hand in

Hand in your answers to the tasks 1 and 3 and the code of class PERSON and feature infect for the task 2.

2 Short trips

In Zurich you can buy a cheaper public transportation ticket if you are doing a short trip (Kurzstrecke). In this task you will develop an application that helps customers decide what type of ticket they need, by visualizing the short-trip range of a given station. We consider a trip short if it takes two minutes or less.
Table 1: Different versions of feature `infect`

<table>
<thead>
<tr>
<th>Version 1</th>
<th>Version 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>infect (p: PERSON)</code></td>
<td><code>infect (p: PERSON)</code></td>
</tr>
<tr>
<td><code>--- Infect ‘p’ and coworkers.</code></td>
<td><code>--- Infect ‘p’ and coworkers.</code></td>
</tr>
<tr>
<td><strong>require</strong></td>
<td><strong>require</strong></td>
</tr>
<tr>
<td><code>p_exists: p /= Void</code></td>
<td><code>p_exists: p /= Void</code></td>
</tr>
<tr>
<td><strong>do</strong></td>
<td><strong>do</strong></td>
</tr>
<tr>
<td><code>if p.coworker /= Void and then</code></td>
<td><code>if p.coworker /= Void and then</code></td>
</tr>
<tr>
<td><code>not p.coworker.has_flu then</code></td>
<td><code>not p.coworker.has_flu then</code></td>
</tr>
<tr>
<td><code>p.coworker.set_flu</code></td>
<td><code>p.coworker.has_flu</code></td>
</tr>
<tr>
<td><code>infect (p.coworker)</code></td>
<td><code>infect (p.coworker)</code></td>
</tr>
<tr>
<td><strong>end</strong></td>
<td><strong>end</strong></td>
</tr>
<tr>
<td><code>p.set_flu</code></td>
<td><code>p.set_flu</code></td>
</tr>
<tr>
<td><strong>end</strong></td>
<td><strong>end</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version 3</th>
<th>Version 4</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>infect (p: PERSON)</code></td>
<td><code>infect (p: PERSON)</code></td>
</tr>
<tr>
<td><code>--- Infect ‘p’ and coworkers.</code></td>
<td><code>--- Infect ‘p’ and coworkers.</code></td>
</tr>
<tr>
<td><strong>require</strong></td>
<td><strong>require</strong></td>
</tr>
<tr>
<td><code>p_exists: p /= Void</code></td>
<td><code>p_exists: p /= Void</code></td>
</tr>
<tr>
<td><strong>local</strong></td>
<td><strong>local</strong></td>
</tr>
<tr>
<td><code>q: PERSON</code></td>
<td><code>q: PERSON</code></td>
</tr>
<tr>
<td><strong>do</strong></td>
<td><strong>do</strong></td>
</tr>
<tr>
<td><code>from</code></td>
<td><code>if p.coworker /= Void and then</code></td>
</tr>
<tr>
<td><code>q := p.coworker</code></td>
<td><code>not p.coworker.has_flu then</code></td>
</tr>
<tr>
<td><code>p.set_flu</code></td>
<td><code>p.coworker.has_flu</code></td>
</tr>
<tr>
<td><strong>until</strong></td>
<td><strong>end</strong></td>
</tr>
<tr>
<td><code>q = Void</code></td>
<td><code>infect (p.coworker)</code></td>
</tr>
<tr>
<td><strong>loop</strong></td>
<td><strong>end</strong></td>
</tr>
<tr>
<td><code>if not q.has_flu then</code></td>
<td></td>
</tr>
<tr>
<td><code>q.set_flu</code></td>
<td></td>
</tr>
<tr>
<td><code>q := q.coworker</code></td>
<td></td>
</tr>
<tr>
<td><strong>end</strong></td>
<td></td>
</tr>
<tr>
<td><strong>end</strong></td>
<td></td>
</tr>
</tbody>
</table>

**To do**


2. Implement a recursive feature `highlight_reachable` that takes two arguments: a station `s` of type `STATION` and a time interval `t` of type `REAL_64`. The feature should highlight all stations that are reachable from `s` in `t` seconds or less. You may use a loop to traverse the lines passing through a given station (accessible through the query `lines`); however you are not allowed to use a loop that traverses all the stations in the city.

**Hint.** We assume that the segment of a public transportation line between any two
adjacent stations is always straight. For that reason you can compute the time it takes to go from a station to the next one, by simply dividing the distance between the station positions by the speed of the line.

3. To test \texttt{highlight\_reachable}, invoke it from the feature \texttt{highlight\_short\_distance} with the time interval of two minutes. The application is programmed to call \texttt{highlight\_short\_distance}, whenever you left-click a station on the map.

\textbf{To hand in}

Hand in the code of \textit{SHORT\_TRIPS}.

3 \textbf{N Queens}

The N-queens problem is the problem of positioning \(N\) queens on an \(N \times N\) chess board such that no queen attacks another (i.e., they do not share the same row, column, or diagonal).

The problem can be solved recursively. For example, Figure 1 shows how a partial solution for the first 4 rows of the board is being extended to the 5\textsuperscript{th} row. The main idea is that if the partial solution is not yet complete, then for each safe location in the current row \(1\), you can add the location to the solution and use this new solution to solve the problem for the next row.

![Figure 1: An example of a partial solution](image)

\textbf{To do}

1. Download \url{http://se.inf.ethz.ch/courses/2012b_fall/eprog/assignments/08/n_queens.zip} unzip it and open \texttt{n\_queens.ecf}. Open class \texttt{PUZZLE}.

2. Implement a recursive procedure \texttt{complete}, which completes a given partial solution. You can make use of a given function \texttt{under\_attack}, which determines if a particular position in the current row is safe; for this function to work correctly you need to implement the helper function \texttt{attack\_each\_other}.

3. Add a call to \texttt{complete} from \texttt{solve}, in such a way that after calling \texttt{solve (n)} the list \texttt{solutions} contains all solutions for the board of size \(n\).

4. Run the program: this will test you implementation on board sizes from 1 to 10. If any of the tests fail, revise your implementation until they pass.

\textbf{To hand in}

Hand in the code of \textit{PUZZLE}.

\footnote{A location is safe if it is not attacked by any of the currently placed queens.