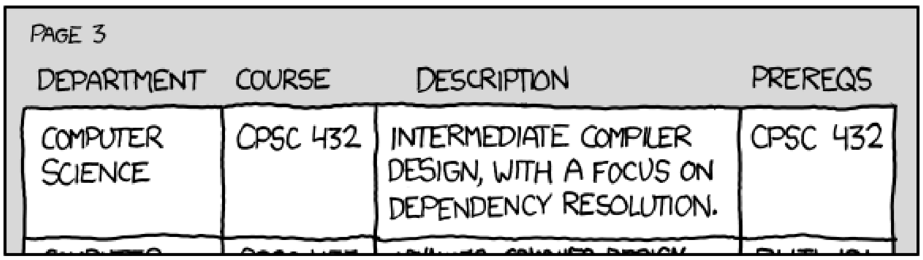
Assignment 8: Recursion

ETH Zurich

Hand-out: 11. November 2011
Due: 22. November 2011



Dependencies © Randall Munroe (http://xkcd.com/754/)

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*

```
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
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*

Version 1

```
infect (p: PERSON)
    -- Infect 'p' and coworkers.
  require
    p_exists: p /= Void
  do
    p.set_flu
    if p.coworker /= Void and then not
         p.coworker.has_flu then
      infect (p.coworker)
    end
  end
```

Version 2

```
infect (p: PERSON)
    -- Infect 'p' and coworkers.
  require
    p_exists: p /= Void
  do
    if p.coworker /= Void and then not
         p.coworker.has_flu then
      infect (p.coworker)
      p.coworker.set_flu
    end
    p.set_flu
  end
```

Version 3

```
infect (p: PERSON)
    -- Infect 'p' and coworkers.
  require
    p_exists: p /= Void
  local
    q: PERSON
  do
    from
      q := p.coworker
      p.set_flu
    until
      q = Void
    loop
      if not q.has_flu then
        q.set_flu
      end
      q := q.coworker
    end
  end
```

Version 4

```
infect (p: PERSON)
    -- Infect 'p' and coworkers.
  require
    p_exists: p /= Void
  do
    if p.coworker /= Void and then
         not p.coworker.has_flu then
      p.coworker.set_flu
      infect (p.coworker)
    end
    p.set_flu
  end
```

To do

1. Download http://se.inf.ethz.ch/courses/2011b_fall/eprog/assignments/08/traffic.zip unzip it and open `assignment_8.ecf`. Open class *SHORT_TRIPS*.

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 *highlight_reachable*, invoke it from the feature *highlight_short_distance* with the time interval of two minutes. The application is programmed to call *highlight_short_distance*, whenever you left-click a station on the map.

To hand in

Hand in the code of *SHORT_TRIPS*.

3 Get me out of this maze!

In this task, you will write an application that reads a maze description from a file and then, given a starting point, calculates a path to an exit. We provide classes for reading the maze files and storing the maze. If you feel adventurous you can also write the entire application yourself (your application should be able to read the maze files provided by us). The main goal, however, is to implement the recursive feature *find_path*.

To do

1. Create a new application in EiffelStudio with a root class *MAZE_APPLICATION*.

2. Download `http://se.inf.ethz.ch/courses/2011b_fall/eprog/assignments/08/maze.zip` and extract it into the project directory. The zip-file contains classes *MAZE_READER* and *MAZE* as well as three maze input files.

   A maze is a rectangular board with width w and height h where each field is either *empty*, a *wall*, or an *exit*.

   Each input file starts with the width and height of the board. They are followed by a map of the maze, where '.' denotes an empty field, '#' denotes a wall, and '*' denotes an exit. Below you see an example 6×6 maze input file. Class *MAZE_READER* reads the file and stores the data in an instance of class *MAZE*.

```
6 6
..####
#....#
#.####
#.#..*
#...##
######
```

3. In the root creation procedure of class *MAZE_APPLICATION* you should ask the user for the name of an input file and use *MAZE_READER* to read the input file into an instance of class *MAZE*. Display the maze on the standard output, then ask the user to input a row and a column number within the maze's dimensions. This will be the starting field for finding a path to an exit. See Figure 1 for an example.

4. In class *MAZE* there is a feature *find_path* whose implementation is missing. The argument of *find_path* defines the starting field. Your implementation should search for a path from the starting field to one of the exits in the maze and store the sequence of moves that are needed to reach it in the attribute *path*. There are four valid moves from a given field:

move one field up (North), move one down (South), move one left (West) and move one right (East). Note that the implementation of *find_path* need not find the shortest path – any path leading to an exit is good enough. In case there is no path leading to an exit, the attribute *path* must remain **Void**.

Figure 1 shows an execution of the system with a maze where a path exists and Figure 2 shows an execution when there is no path.

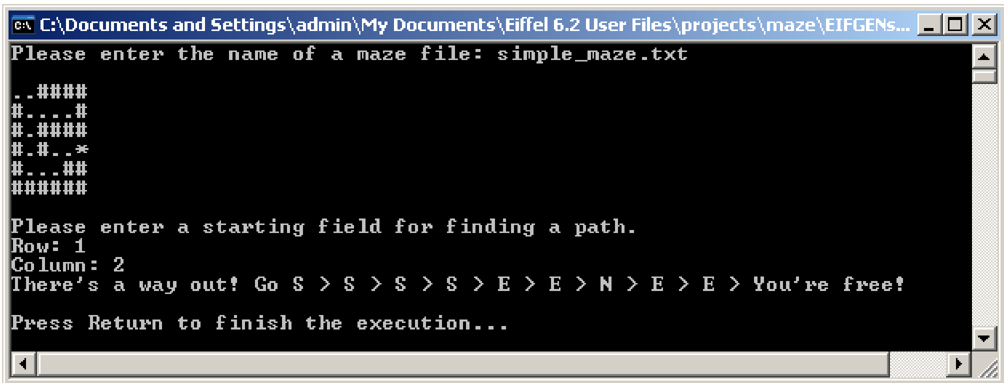


Figure 1: Maze with a path.

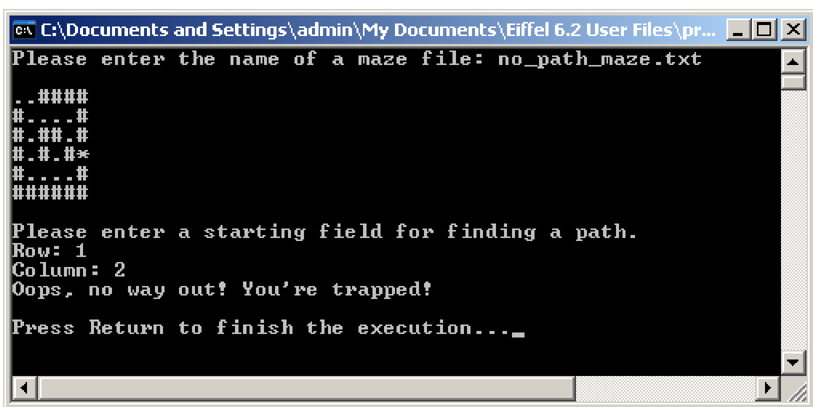


Figure 2: Maze with no path.

To hand in

Hand in the source code of your application.