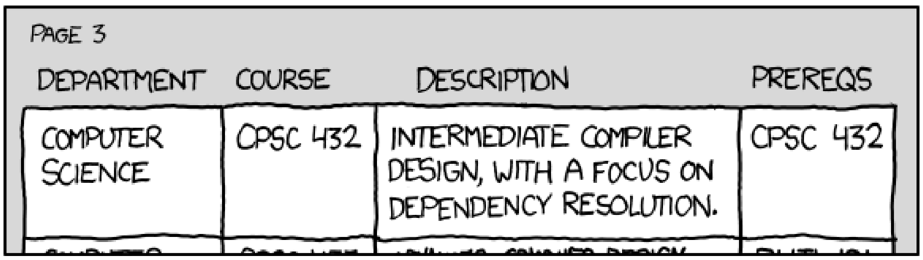
Assignment 9: Recursion

ETH Zurich

Hand-out: 19. November 2010
Due: 30. November 2010



Dependencies © Randall Munroe (xkcd.com)

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 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. The coworker relation is asymmetric.

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 Reachable stations

In this task you will write a procedure that, given a station, highlights all stations that are reachable from it within a certain time limit (e.g. 10 minutes).

Figure 1 shows an example of stations and stops in Traffic. Every *TRAFFIC_STATION* contains a list of *TRAFFIC_STOP*s. Every *TRAFFIC_STOP* represents a stop of a certain

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
```

TRAFFIC_LINE in one direction. For example, the lowest stop of station Haldenegg in Figure 1 represents the stop for Tram 7 coming from Sonneggstrasse and continuing towards Central.

Class *TRAFFIC_STOP* offers the following useful features:

- *station* – the associated station;
- *right* – the next stop;
- *time_to_next* – time (in minutes) it takes to travel from the stop to the next stop.

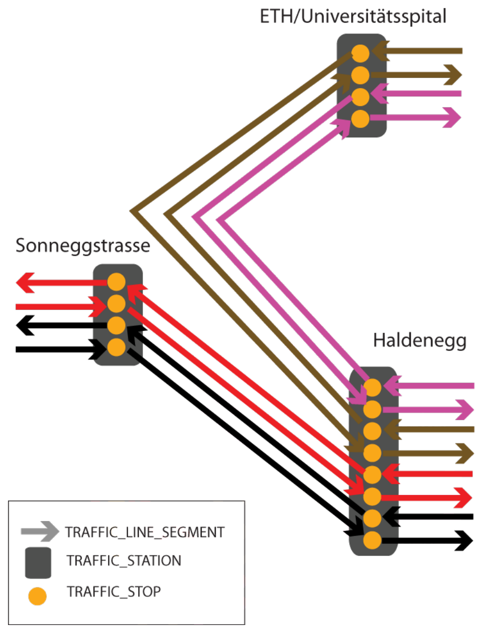


Figure 1: Example stops, stations, and segments.

To do

1. Download http://se.ethz.ch/teaching/2010-H/eprog-0001/assignments/09/assignment_9.zip and extract it in `traffic/example`. You should now have a new directory `traffic/example/assignment_9` with assignment_9.ecf directly in it.

2. Open and compile this new project and navigate to class *RECURSIVE_HIGHLIGHTING*.

3. Implement a recursive feature *highlight_reachable_stations* that takes two arguments: a station *s* of type *TRAFFIC_STATION* and a time *t* of type *REAL_64*. The feature should highlight all stations that are reachable from *s* in less time than *t* minutes. You may use a loop to iterate through the stops of a certain station (accessible through the query *stops*).

   Test your implementation of *highlight_reachable_stations* with some of the predefined stations of Paris (such as *Station_chatelet* or *Station_Invalides*) and a certain time limit such as 10 minutes.

To hand in

Hand in the code of *RECURSIVE_HIGHLIGHTING*.

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* and a creation feature *make*.

2. Download http://se.ethz.ch/teaching/2010-H/eprog-0001/assignments/09/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 feature *make* 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 read maze in the console. 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 2 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. 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* does not need to find the shortest path – any path leading to an exit is good enough. The feature *find_path* should also set *path_exists* to **True** if a path is found, and to **False** if there is no way out of the maze. Figure 2 shows an execution of the system with a maze where a path exists and Figure 3 shows an execution when there is no path.

Hint

You can base the algorithm on the following idea: to find a path from a certain position on the board make a step in one of the possible directions to a field that has not yet been explored and then try finding your way from there.

To hand in

Hand in the source code of your application.

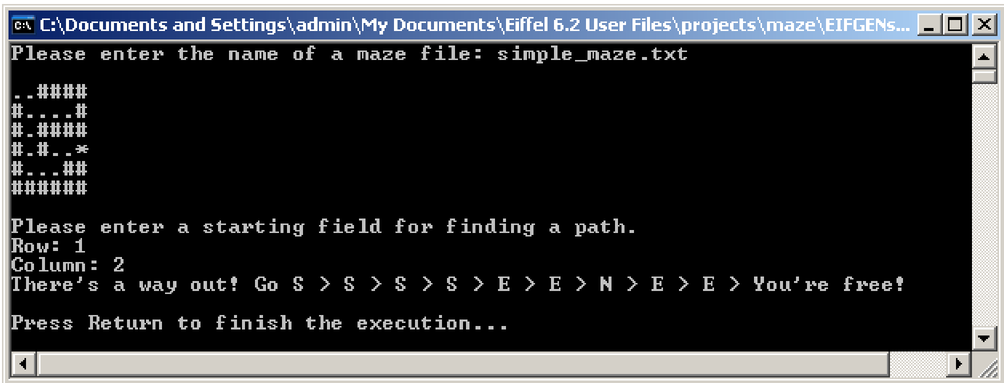


Figure 2: Maze with a path.

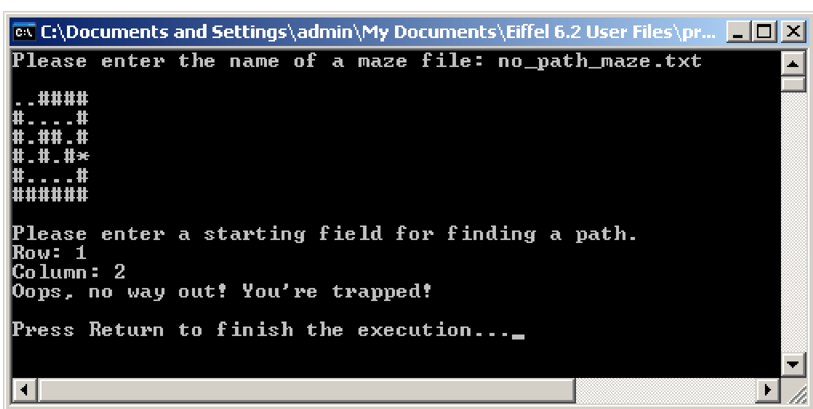


Figure 3: Maze with no path.