

# Assignment 9: Recursion

ETH Zurich

Hand-out: 21. November 2008  
Due: 27. November 2008

## Goals

- Test your understanding of recursion.
- Implement recursive algorithms.

## 1 Propagating the flu

The following class *PERSON* models coworkers and class *APPLICATION* creates *PERSON* objects and sets up the coworker structure. Note that the setup is somewhat superficial since the coworker relation is asymmetric and restricted to at most one person.

Assume you are the boss of these people and you want to make the following inference: if a person *p* has the flu, then *p* spreads the infection to the person he or she has as coworker, who then spreads it to his/her coworker, and so on.

Listing 1: Class *PERSON*

```
1 class PERSON
  create make
3 feature -- Initialization

5   make (a_name: STRING)
      -- Initialize with name.
7   require
      a_name_valid: a_name /= Void and then not a_name.is_empty
9   do
      name := a_name
11  ensure
      name_set: name = a_name
13  end

15 feature -- Access

17  name: STRING -- First name

19  coworker: PERSON -- Person that Current works with

21  has_flu: BOOLEAN -- Does he/she have the flu?

23 feature -- Element change
```

```
25 set_coworker (p: PERSON)
    -- Set 'coworker' to 'p'.
27 require
    p_exists: p /= Void
29    p_different: p /= Current
    do
31        coworker := p
    ensure
33        coworker_set: coworker = p
    end
35
    set_has_flu
37    -- Set 'has_flu' to True.
    do
39        has_flu := True
    ensure
41        has_flu: has_flu
    end
43
invariant
45 name_valid: name /= Void and then not name.is_empty
end
```

Listing 2: Class *APPLICATION*

```
class APPLICATION
2 create make
  feature -- Initialization
4
    make is
6        -- Run application.
        local
8            joe, mary, tim, sarah, adam, bill, cara: PERSON
        do
10            create joe.make ("Joe")
            create mary.make ("Mary")
12            create tim.make ("Tim")
            create sarah.make ("Sarah")
14            create bill.make ("Bill")
            create cara.make ("Cara")
16            create adam.make ("Adam")
            joe.set_coworker (sarah)
18            adam.set_coworker (joe)
            tim.set_coworker (sarah)
20            sarah.set_coworker (cara)
            bill.set_coworker (tim)
22            cara.set_coworker (mary)
            mary.set_coworker (bill)
24            infect (bill)
        end
26
end
```

You are now being provided with several solutions to the problem, all called *infect* (see Variant 1 to Variant 4).

Variant 1 of *infect*

```

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

Variant 2 of *infect*

```

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

Variant 3 of *infect*

```

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

Variant 4 of *infect*

```

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

## To do

1. Consider all the variants in turn and say
  - whether or not the solution does what you want it to do
  - if it does, say how it does it (in one to two sentences in your own words), or, if it doesn't, say why it doesn't (again in your own words).

Note that this is a pen-and-paper task - you do not need EiffelStudio to solve this.

2. **Optional:** The class *PERSON* above lets each person only have one coworker. This is rather unusual. Rewrite the class *PERSON* such that a person can have as many different

coworkers as he/she wants. Then implement a correct recursive feature *infect* for this new situation. Note: you may use a loop to iterate through the list of coworkers.

### To hand in

Submit your answers to your assistant.

## 2 Calculating reachable stations in Paris

In this task, you will write a recursive procedure that given a station in Paris recursively calculates all stations are reachable within a certain time limit (e.g. 10 minutes) and highlights them on the map.

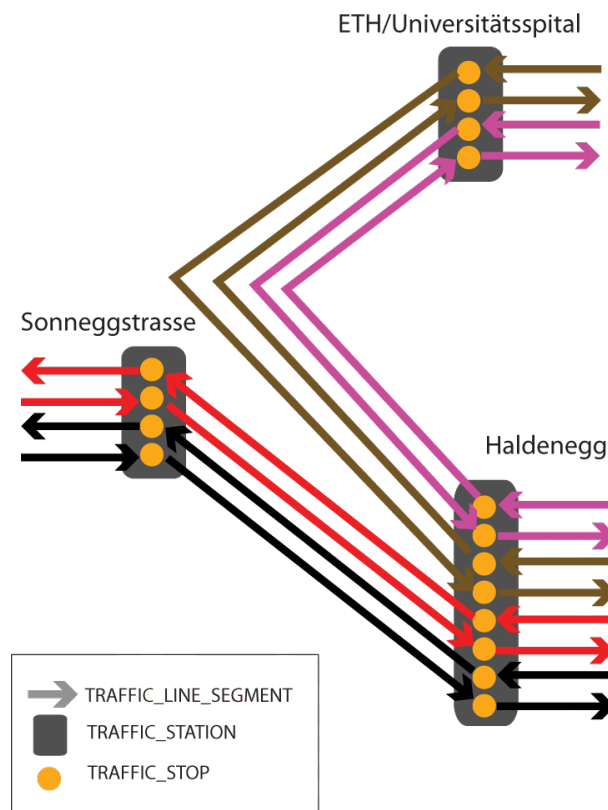


Figure 1: Example stops, stations, and segments.

For this, you need to understand the interplay of stations and stops. Figure 1 shows a schematic overview on the example of the stations Haldenegg, Sonneggstrasse and ETH/Universitätsspital in Zurich. Every *TRAFFIC\_STATION* (dark grey rounded rectangles in Figure 1) has a set of *TRAFFIC\_STOPS* (orange circles) that are associated with it. Every *TRAFFIC\_STOP* represents a stop of a certain *TRAFFIC\_LINE* in one direction. For example, the lowest stop of station Haldenegg in Figure 1 represents the stop for Tram number 7 coming from Sonneggstrasse and continuing towards Central while the second lowest stop represents a stop of the same line in the opposite direction.

Class *TRAFFIC\_STATION* offers a feature *highlight* that will set *is\_highlighted* to *True* and update the display of a station. It also offers a feature *stops* that returns a list of its associated

stops of type *TRAFFIC\_STOP*. The class *TRAFFIC\_STOP* has a feature *station* giving access to its associated station, a feature *right* that returns the next stop (for the second lowest stop of Haldenegg it would for example return the second lowest stop of station Sonneggstrasse), and a feature *time.to.next* that calculates the time in minutes it takes to travel from the stop to its next stop (as returned by feature *right*).

## To do

1. Download [http://se.ethz.ch/teaching/2008-H/eprog-0001/exercises/assignment\\_9.zip](http://se.ethz.ch/teaching/2008-H/eprog-0001/exercises/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 (it is important that the location corresponds to the description here!).
2. Open and compile this new project and navigate to class *RECURSIVE\_HIGHLIGHTING*.
3. Implement a recursive feature called *highlight\_reachable\_stations* that takes two arguments: a station *s* of type *TRAFFIC\_STATION* and a time *t* of type *REAL*. The feature should highlight all stations that are reachable from *s* in less time than *t* minutes. Test your implementation of *highlight\_reachable\_stations* with some of the predefined stations of Paris (such as *Station\_balard* or *Station\_Invalides*) and a certain time limit such as 10 minutes.

## Hints

- The feature *stops* of *TRAFFIC\_STATION* lets you access all directly connected stations since each of the associated stops provides its next stop through feature *right*.
- You may use a loop to iterate through the stops of a station.

## To hand in

Hand in class *RECURSIVE\_HIGHLIGHTING*.

### 3 Get me out of this maze!

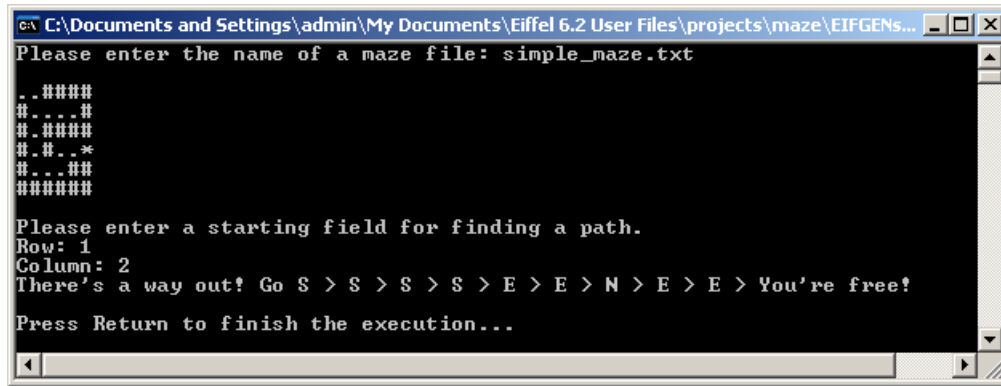
In this task, you will write an application that reads a maze from a file and then, given a starting point, calculates a path to an exit. We provide classes for reading the maze files and storing the data in an appropriate data structure, but if you feel adventurous, you can also write the entire application yourself including the parser for the maze files and the class that stores the maze. Note that the main goal of this task is to write the recursive feature *find\_path*, so please only choose to write the entire application yourself, if you are certain that you will not get caught up with writing the parser. Whatever you choose, the output of your application should look similar to the example outputs shown in Figure 2 and Figure 3 and you should be able to read the maze files provided in the downloadable zip and provide a working feature that finds a path out of the maze. The description below targets those that choose to use our files, but also contains information for those that choose to write the entire application themselves.

#### To do

1. Create a new application in EiffelStudio with a root class *MAZE\_APPLICATION* and creation feature *make*.
2. Download <http://se.ethz.ch/teaching/2008-H/eprog-0001/exercises/maze.zip> and put the extracted files in the project directory. The zip-file contains files for a class *MAZE\_READER* and a class *MAZE* and 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*. In the example files, the first number defines the width of the board and the second number defines the height of it. Every appearance of the character '.' denotes an empty field, '#' a wall, and '\*' an exit. Below you see an example maze input file with dimension 6 x 6. 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* let the user input a file name and use *MAZE\_READER* to read it into an instance of class *MAZE*. Display the read maze in the console. Then ask the user to input a row and a column 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 and 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. *find\_path* should also set *path\_exists* to True if a path is found, or false if there is no path 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 exists no path.

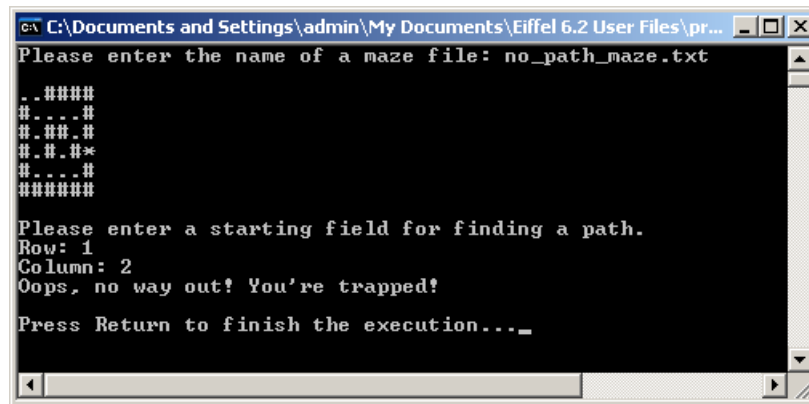


```
C:\Documents and Settings\admin\My Documents\Eiffel 6.2 User Files\projects\maze\EIFGENs...
Please enter the name of a maze file: simple_maze.txt
..####
#. . . #
#. ####
#. # . *
#. . . #
#####

Please enter a starting field for finding a path.
Row: 1
Column: 2
There's a way out! Go S > S > S > S > E > E > N > E > E > You're free!

Press Return to finish the execution...
```

Figure 2: Maze with a path.



```
C:\Documents and Settings\admin\My Documents\Eiffel 6.2 User Files\pr...
Please enter the name of a maze file: no_path_maze.txt
..####
#. . . #
#. ####
#. # . *
#. . . #
#####

Please enter a starting field for finding a path.
Row: 1
Column: 2
Oops, no way out! You're trapped!

Press Return to finish the execution..._
```

Figure 3: Maze with no path.

## Hints

- The *find\_path* implementation of the master solution uses a backtracking algorithm that starts trying all possible ways through the maze and stops as soon as it has a path. It is based on the following idea: to find a path from a certain position (i, j) on the board, unless it is already the exit (base case), you need to move in one of the four directions (north, south, west and east). If the step would move you out of the board, this is an invalid move. If you've already been at the position then you don't have to explore further. If the field is a wall then you can't move there. In all these cases, you try the next available move for (i, j). If the field you want to move to is an empty field, you can move there and try to find the path from there.
- If you choose to write the parser yourself, you will find the class [PLAIN\\_TEXT\\_FILE](#) helpful.

## To hand in

Hand in the source code of your application.