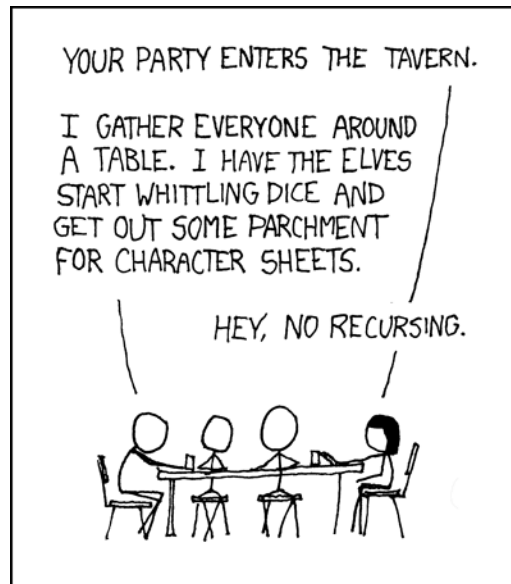


Assignment 9: Recursion

ETH Zurich

Hand-out: 20. November 2009
Due: 1. December 2009



Copyright Randall Munroe <http://xkcd.com>

Goals

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

1 An infectious task

You are the boss of a company following with anxiety the news about the flu pandemic. To take a better decision about the actions to take, you decide to simulate the spreading of the flu in a program. For this you assume the following model: if a person p has the flu, then p 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 and restricted to at most one person, in order to reflect the assumption above.

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
6   -- Run application.
  local
8   joe, mary, tim, sarah, bill, cara, adam: PERSON
  do
10  create joe.make ("Joe")
11  create mary.make ("Mary")
12  create tim.make ("Tim")
13  create sarah.make ("Sarah")
14  create bill.make ("Bill")
15  create cara.make ("Cara")
16  create adam.make ("Adam")
17  joe.set_coworker (sarah)
18  adam.set_coworker (joe)
19  tim.set_coworker (sarah)
20  sarah.set_coworker (cara)
21  bill.set_coworker (tim)
22  cara.set_coworker (mary)
23  mary.set_coworker (bill)
24  infect (bill)
  end
26
end
```

You are now being provided with four implementations of feature *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
```

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
```

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
```

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
```

To do

1. Consider all the variants in turn and write down
 - Whether or not the solution does what it is supposed to do
 - If it does, explain how in your own words (one to two sentences)
 - If it doesn't, explain why in your own words (one to two sentences)

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

2. **Optional for experience level A, suggested for experience levels B and C students:** The class *PERSON* above assumes that each person can only infect one coworker. This is unfortunately too optimistic. Rewrite the class *PERSON* such that a person can have (and infect) many different coworkers. 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 procedure that given a station in Paris does the following:

- Recursively finds all stations that are reachable within a certain time limit (e.g. 10 minutes)
- Highlights the found stations on the map

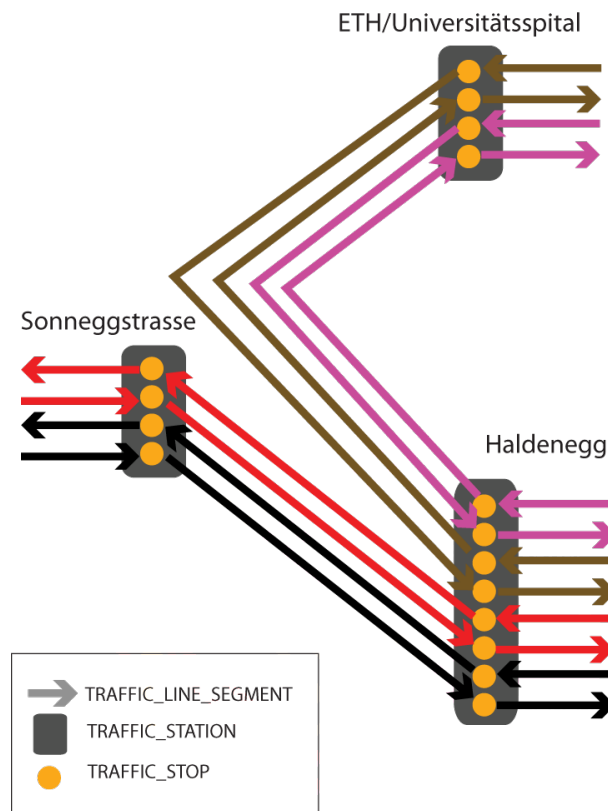


Figure 1: Example stops, stations, and segments.

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 the following useful features:

- *highlight* will set *is_highlighted* to *True* and update the display of a station

- *stops* returns a list of its associated stops of type *TRAFFIC_STOP*

Class *TRAFFIC_STOP* offers the following useful features:

- *station* gives access to its associated station
- *right* returns the next stop. For the second lowest stop of Haldenegg it would for example return the second lowest stop of station Sonneggstrasse
- *time_to_next* calculates the time it takes to travel from the stop to its next stop, as returned by feature *right*. The time is expressed in minutes

To do

1. Download http://se.ethz.ch/teaching/2009-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. If you feel adventurous, or if you are in a B or C experience level group, 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*. 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, 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 output of your application should look similar to the example outputs shown in Figure 2 and Figure 3. 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.

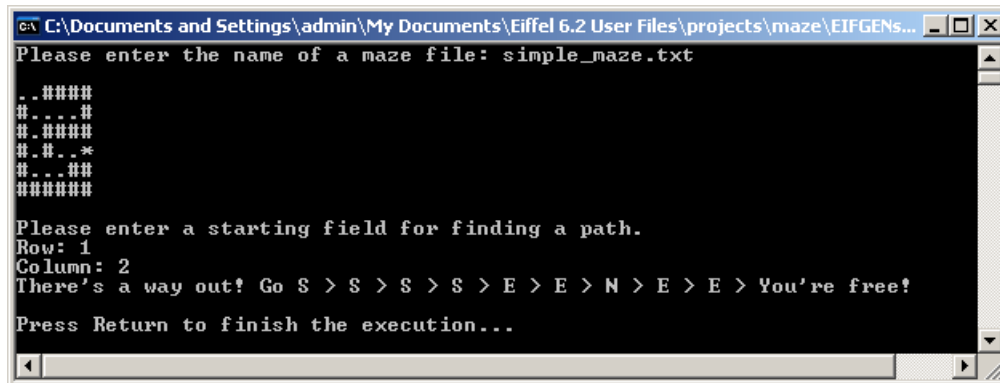
To do

1. Create a new application in EiffelStudio with a root class *MAZE_APPLICATION* and a creation feature *make*.
2. Download <http://se.ethz.ch/teaching/2009-H/eprog-0001/exercises/maze.zip> and put the extracted files into the project directory. The zip-file contains the files for the class *MAZE_READER* and the 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* you should ask the user for the name of an input a 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 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, 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.

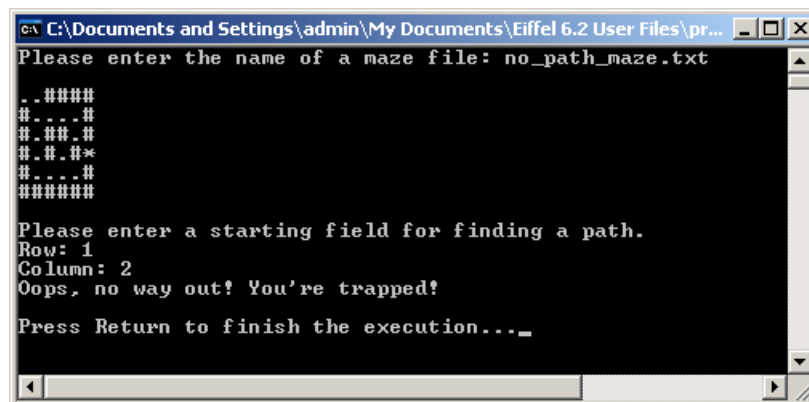


```
C:\Documents and Settings\admin\My Documents\Eiffel 6.2 User Files\projects\maze\EIFGENs...
Please enter the name of a maze file: simple_maze.txt
..####
#...#
#.#.#
#.#.*
#...#
#####

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 an algorithm based on the following idea: to find a path from a certain position (i, j) on the board (excluding the exit), you need to move in one of the four directions (north, south, west or east). You should try all the valid moves that have not been tried before. Invalid moves are the ones that put you out of the board or on a wall
- 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.