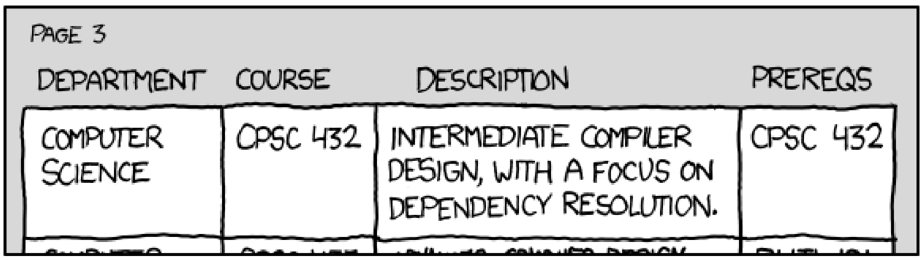
Assignment 8: Recursion

ETH Zurich

Handout: 9. November 2015
Due: 18. November 2015



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 the health of your employees. Winter is coming, and with it the usual flu epidemics, not to mention the Ebola virus concern you have been reading about lately. 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 (written sentences) to tasks 1 and 3 and the code of class *PERSON* and feature *infect* for task 2.

Table 1: Different versions of feature *infect*

Version 1

```
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
```

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
        p.set_flu
        if p.coworker /= Void and then not
                p.coworker.has_flu then
            infect (p.coworker)
        end
    end
```

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.

To do

1. Download `http://se.inf.ethz.ch/courses/2015b_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 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^{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.

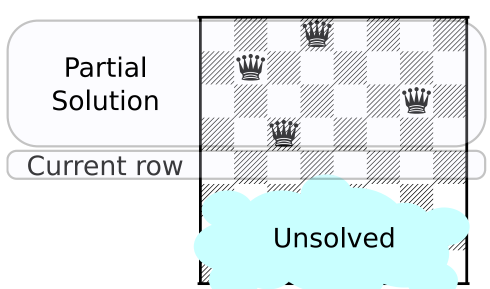


Figure 1: An example of a partial solution

To do

1. Download `http://se.inf.ethz.ch/courses/2015b_fall/eprog/assignments/08/n_queens.zip` unzip it and open `n_queens.ecf`. Open class *PUZZLE*.

[1] A location is safe if it is not attacked by any of the currently placed queens.

2. Implement a recursive procedure *complete*, which completes a given partial solution. You can make use of a given function *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 *attack_each_other*.
3. Add a call to *complete* from *solve*, in such a way that after calling *solve* (n) the list *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.

To hand in

Hand in the code of *PUZZLE*.

4 MOOC: Design by Contract, recursion

To do

1. Access the main MOOC course web page at `http://se.ethz.ch/mooc/programming`.
2. Listen to lecture number 10 "Design by Contract" and take the corresponding quizzes.
3. Listen to lecture number 13 "Recursion", take the corresponding quiz and solve the programming exercise.

Your goal is to provide all correct answers to the quizzes. You can take the quizzes as many times as you want. If you succeed, you will be awarded a badge for each correctly solved quiz.