CS2104 – Programming Languages
School of Computing
National University of Singapore

September 18, 2012

Problem Set 4

Semester 1, 2012/13

Due: ***October 7, 23:59***

Marks: 16

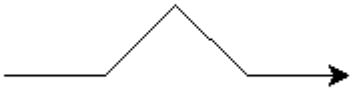
Submission: In IVLE, in the cs2104 workbin, you will find a folder called "`Homework submissions`". In that folder, there are currently 4 ***subfolders:*** **`PS4P01, ..., PS4P04`**. The last two digits of the folder name indicate the solution that is to be submitted into that folder: the solution to *Question 1* into **`PS4P01`**, and so on (that is, you need to submit 4 separate solutions to 4 problems). A solution should consist of a ***single text file*** that can be compiled, or loaded into the interpreter of interest and executed. You should provide as much supplementary information about your solution as you can, ***in the form of program comments***. Moreover, if you work in a team, state the members of the team at the beginning of the file, in a comment. You do not need to submit the same file twice, one submission per team is sufficient.

Problem 1 [C][4 marks, submit to `PS4P01`]

This problem is concerned with drawing fractals in Python. However, it is, essentially, a language translation problem. The first step in solving this problem is getting yourselves familiarized with *the turtle graphics package in Python*. The full documentation is available at http://docs.python.org/py3k/library/turtle.html. However, we need a very small part of this library to carry out this assignment. Open you Python shell, and type the following program snippet:

```
from turtle import *
delay(0)
forward(50)
left(45)
forward(50)
right(90)
forward(50)
left(45)
forward(50)
```

This little snippet will open a separate window, and draw the following graph in it:



It's probably pretty clear what the commands are doing. The first line imports the turtle graphics library. The `delay` command sets the delay between successive updates on the screen and will be useful in making Python draw faster later (currently, speed is not an issue, but it will soon become). The forward command moves along the current direction for a distance equal to its argument. The commands left and right change the current direction by the specified angle. While the turtle library has many more commands, the commands given above are the only ones you need to learn for this assignment.

Towards a fractal description language

We shall now move to the Prolog environment again. We propose the following graphics description language:

- `unit` specifies the drawing of a "unit" segment. The length of the unit segment will be specified as an argument to predicates that are described later.
- `left(Angle)` changes the current direction to the left by the specified angle, in degrees.
- `right(Angle)` changes the current direction to the right by the specified angle, in degrees.
- `Image1;Image2` sequences the drawing of two images: `Image1` and `Image2`. First `Image1` must be drawn, and then, keeping the current direction as it was left after drawing `Image1`, `Image2` must be drawn.

For example, the image above would be described by the following graphics expression:

```
unit; left(45);unit;right(90);unit;left(45);unit
```

In this language, we shall regard the semicolon as associative. Thus, the following expression is equivalent to the one above (pay attention to the brackets):

```
(unit;left(45);unit); right(90);unit;left(45);unit
```

A useful exercise at this point (though not a hard requirement of this problem) is to write a Prolog predicate that translates a graphics description expression into the sequence of Python turtle graphics commands that draw the figure. The predicate should take as the first argument the graph expression that you want drawn, and as the second argument, the length of the unit segment. Note that you do not have to run Python from Prolog. All you need to do is print the commands on the screen, so they can be copied and pasted into the Python environment and executed.

Fractals

For a down-to-earth tutorial on fractals, you may read
http://library.thinkquest.org/26242/full/tutorial/tutorial.html.

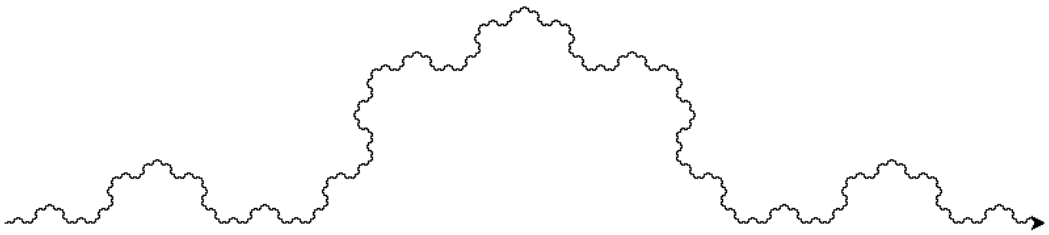
We shall be concerned with the simplest types of fractals, the ones that are *perfectly self-similar*. First, let us notice that our graphic expression language can be used as a graph transformation language. Let us assume that `g` is a graph that can be drawn by a sequence of turtle graphics commands. Then an expression of the sort

```
g; left(45); g; right(90); g; left(45); g
```

is a new graph, which can be construed as a function of the argument `g`. Now, a fractal is the graph that represents a solution to a fix-point equation of the form `g = F(g)`, where `F` is a graph drawing expression. For instance, the equation:

```
g = (g; left(45);g;right(90);g;left(45);g)
```

defines a fractal, whose graph is the following:

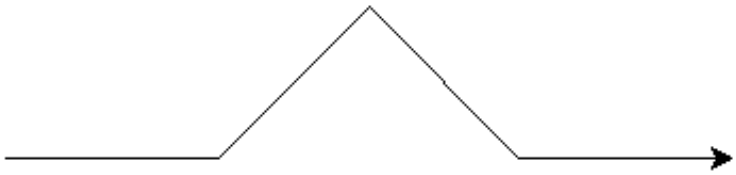


(this is, in fact, an approximation of the actual solution). The important aspect here is that, if we denote `F(x) = (x; left(45);x;right(90);x;left(45);x)`, then `F(g)` is the same as `g` (and in that sense, `g` is ***self-similar***).

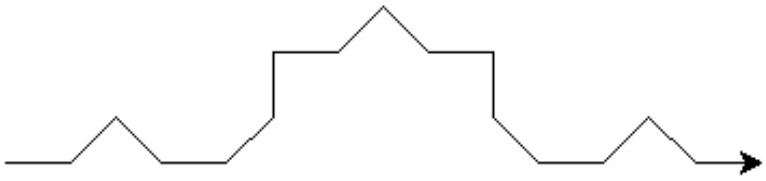
Now, let us define a fractal approximation. Given a fractal `g=F(g)`:

- The approximation of level 0 is `unit`
- The approximation of level `k` is $F(g^{k-1})$, where g^{k-1} denotes the approximation of level `k-1`.

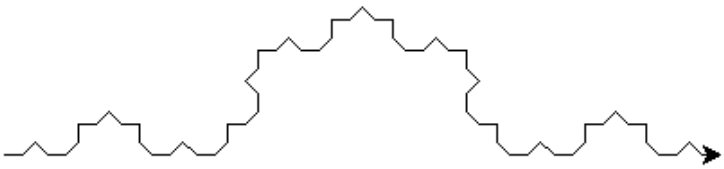
For our example, here are the first 5 approximations:



k = 1

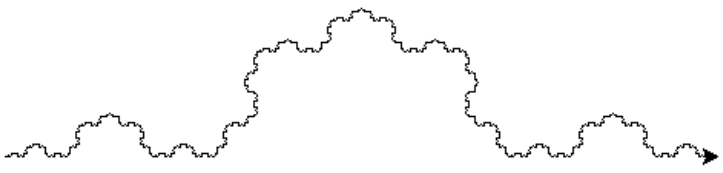


k = 2



k = 3

k = 4



k = 5 (almost identical to k=4)

Now, here's the actual task of this problem. Write a Prolog predicate called fractal, which takes in the following arguments:

- A fractal equation
- An integer representing an approximation level
- A real number representing the length of the unit segment

This predicate should generate as written output (that is, output should be generated using the `write` predicate) the sequence of turtle commands that draw the corresponding fractal approximation. For instance, the query:

```
?-  fractal(g = (g;left(45);g;right(90);g;left(45);g),3,30).
```

should produce the output:

```
from turtle import *
delay(0)
forward(30)
left(45)
forward(30)
right(90)
forward(30)
left(45)
forward(30)
left(45)
forward(30)
left(45)
forward(30)
right(90)
forward(30)
left(45)
forward(30)
right(90)
forward(30)
left(45)
forward(30)
right(90)
forward(30)
left(45)
forward(30)
left(45)
forward(30)
left(45)
forward(30)
right(90)
forward(30)
left(45)
forward(30)
```

Note that the default delay is 10ms, and if not changed, will slow down quite a bit the drawing of your graph. Adding `delay(0)` to your program will improve the speed . It may also help to use the Prolog predicates `tell(Filename)` and `told`, to have the output redirected into a file, since the output may be quite large, and difficult to copy-and-paste. Use `help(tell)` and `help(told)` to learn about these predicates. The redirected output can be placed directly into a .py file that can be double-clicked to be executed. If you do that, you may also want to add `time.sleep(some_interval)` at the end of your generated Python program, so that the graph doesn't disappear from the screen right after the drawing has finished.

In testing your predicate, you may want to try out the following fractals:

```
g=(left(120);g;right(60);g;right(60);g;right(60);g;g;left(60))
g=(left(30);g;right(120);g;left(120);g;right(30))
```

Problem 2 [C][4 marks, submit to PS4P02]

Instead of generating a lengthy sequence of turtle graphics commands for Problem 1, generate a short, recursive Python function that achieves the same graphic output. For instance, the family of fractal approximations used as an example in Problem 1 can be generated by the following Python program:

```
from turtle import *
delay(0)
def g(level,unit) :
    if level == 0 :
        forward(unit)
    else:
        g(level-1,unit)
        left(45)
        g(level-1,unit)
        right(90)
        g(level-1,unit)
        left(45)
        g(level-1,unit)

g(3,30) # the empty line above this call is significant!
```

Problem 3 [A][4 marks, submit to PS4P03]

Python has a type of statement called *simultaneous assignment.* It is in fact a consequence of Python's way of handling complex datatype assignments. Thus, an assignment of the form:

```
(x,y) = (x+y,x-y)
```

would first compute the values of `x+y` and `x-y`, and then (i.e. only after both expressions have been computed), the results will be assigned to variables `x` and `y`. This makes it possible to use the following statement for exchanging the values of two variables:

```
(x,y) = (y,x)
```

Modify the compiler given in `compiler.pro` (discussed in Lecture 5) so that it can accept and compile correctly this type of assignment. You may assume that the tuples of variables and expressions never have more than 2 elements.

Problem 4 [4 marks, submit to PS4P04]

Consider augmenting the toy language discussed in Lecture 5 with `for` loops. They would have the syntax:

```
for(Stmt1;Stmt2;Stmt3) do { Stmt }
```

Augment the compiler rules given in the Prolog program `compiler.pro` so that these new statements can be correctly compiled. Test your compiler on the toy program:

```
s=0 ; for(i=0;i<10;i=i+1) do { s = s + i }
```

By compiling and running the resulting VAL program, the following output should be produced:

```
i=10
s=45
```