Homeworks 1

Louis Carnec
15204934
September 22, 2015

Exercise 0.8

Unary multiplication by repeated addition: copy the ones from the second argument to the end of the tape and repeat this process for each of the 1s in the first argument. See note in the code.

```
def turing(code, tape, initPos = 0, initState = "initial"):
  position = initPos
  state = initState
  while state != "halt":
    print state, ": position", position, "in", tape
    symbol = tape[position]
    (symbol, direction, state) = code[state][symbol]
    if symbol != "noWrite": tape[position] = symbol
    position += direction

#This code marks the start of both unary numbers using A and B for arguments
#1 and 2 respectively.
code = {}

code["initial"] = {}
code["initial"][" "] = (" ", 1, "goRightA")

code["goRightA"] = {}
code["goRightA"]["1"] = ("A", 1, "goRighttimes")

code["goRighttimes"] = {}
code["goRighttimes"]["1"] = ("1", 1, "goRighttimes")
code["goRighttimes"][" "] = ("*", 1, "arg2")
```

```
code["arg2"] = {}
code["arg2"]["1"] = ("B", 1, "goRight2")

code["goRight2"] = {}
code["goRight2"]["1"] = ("1", 1, "goRight2")
code["goRight2"][" "] = ("E", 1, "newArg")

#A new arg is created by writing 1 following the blank after the second arg,
#where the answer to the unary multiplication will be printed.
code["newArg"] = {}
code["newArg"][" "] = ("1", 1, "printed1")

code["printed1"] = {}
code["printed1"][" "] = (" ", -1, "toStart")

#We then return to the start.
code["toStart"] = {}
code["toStart"]["1"] = ("1", -1, "toStart")
code["toStart"]["X"] = ("X", -1, "toStart")
code["toStart"]["*"] = ("*", -1, "toStart")
code["toStart"]["B"] = ("B", -1, "toStart")
code["toStart"]["E"] = ("E", -1, "toStart")
code["toStart"]["Y"] = ("Y", 1, "pass1")
code["toStart"]["A"] = ("A", 1, "pass1")
code["toStart"][" "] = (" ", 1, "testArg1")

#print X to iterate over arg 1
code["pass1"] = {}
code["pass1"][" "] = (" ", 1, "finish")
code["pass1"]["A"] = ("X", 1, "toArg2")
code["pass1"]["X"] = ("X", 1, "pass1")
code["pass1"]["1"] = ("X", 1, "toArg2")

#skip to arg 2
code["toArg2"] = {}
code["toArg2"]["1"] = ("1", 1, "toArg2")
code["toArg2"]["*"] = ("*", 1, "markArg2")

#mark argument 1
code["markArg2"] = {}
```

```
code["markArg2"]["B"] = ("Y", 1, "gotoPrint")
code["markArg2"]["Y"] = ("Y", 1, "markArg2")
code["markArg2"]["1"] = ("Y", 1, "gotoPrint")

#after marking arg 2 we print 1 in the new arg and back up.
code["gotoPrint"] = {}
code["gotoPrint"]["1"] = ("1", 1, "gotoPrint")
code["gotoPrint"]["E"] = ("E", 1, "gotoPrint")
code["gotoPrint"][" "] = ("1", -1, "backingUp")

#backing up over arg 2.
code["backingUp"] = {}
code["backingUp"]["1"] = ("1", -1, "backingUp")
code["backingUp"]["E"] = ("E", -1, "backingUp")
code["backingUp"]["Y"] = ("Y", -1, "backingup")
code["backingUp"]["*"] = ("*", 1, "testArg2")

#testing to see if arg 2 has been fully marked. if not go back to marking.
code["testArg2"] = {}
code["testArg2"]["Y"] = ("Y", 1, "testArg2")
code["testArg2"]["1"] = ("Y", 1, "gotoPrint")
code["testArg2"]["E"] = ("E", -1, "eraseMarks2")

#erase the Ys.
code["eraseMarks2"] = {}
code["eraseMarks2"]["Y"] = ("1", -1, "eraseMarks2")
code["eraseMarks2"]["*"] = ("*", -1, "toStart")

#testing to see if arg 1 is fully market. if not go back to marking.
code["testArg1"] = {}
code["testArg1"][" "] = (" ", 1, "testArg1")
code["testArg1"]["X"] = ("X", 1, "testArg1")
code["testArg1"]["1"] = ("X", 1, "toArg2")
code["testArg1"]["*"] = ("*", -1, "eraseMark1")

#erase marks from arg 1
code["eraseMark1"] = {}
code["eraseMark1"]["X"] = ("1", -1, "eraseMark1")
code["eraseMark1"]["1"] = ("1", -1, "eraseMark1")
code["eraseMark1"][" "] = (" ", 1, "finishingUp")
```

```
#finishing up by getting rid of symbols
code["finishingUp"] = {}
code["finishingUp"][" "] = (" ", 1, "finishingUp")
code["finishingUp"]["1"] = ("1", 1, "finishingUp")
code["finishingUp"]["*"] = (" ", 1, "finishingUp")
code["finishingUp"]["E"] = (" ", -1, "finished")

code["finished"] = {}
code["finished"]["1"] = ("1", -1, "finished")
code["finished"][" "] = (" ", -1, "done")

code["done"] = {}
code["done"]["1"] = ("1", -1, "done")
code["done"][" "] = ("1", 0, "halt")

#added blank spaces because list is finite.
turing(code, [" ", "1", "1", "1", " ", "1", "1", "1", "1", " ", " ", " "," ",
" ", " "," ", " ", " "," ", " ", " "," ", " ", " "], 0, "initial")
```

Exercise 0.9

This code is the implementation of a BSS machine using Newton's method for computing the square root of a real number S.

We have

$$x^2 - S = 0 \tag{1}$$

and thus,

$$S = x^2$$

$$\sqrt{s} = x$$

Applying Newton's method to equation (1), we have equation (2) which will be implemented through the Blum Smale and Shub machine.

$$x_{n+1} = (1/2)(x_n + S/x_n) \tag{2}$$

The approximation of x_0 will give us an approximated value for $\sqrt{S}$.

The machine takes an input register of the form:

$$[0.0, 1.9, 3.2, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 2.0, 0.0004]$$

where each of the register's index is associated some variable (see Table 1).

Position i,j		0	1	2	3	4	5	6	7	8	9	10	11	12
Input		0.0	1.9	3.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	2.0	0.0004
Variable	Outgoing	x_n	S	0.0	0.0	0.0	$(x_{n+1} - x_n)^2$	$\frac{x_n + S/x_n}{2}$	$x_n + S/x_n$	S/x_n	c_1	c_2		$\|\varepsilon\|$

Table 1: Input register for a BSS machine

How the machine works:

1. The user inputs constants into the register.
2. Equation (2) is then calculated by telling the machine to conduct some arithmetic operation by computing which inputs to add, subtract, multiply or divide with one another/or themselves.
3. The result of the operation is then printed in register[0] (index 0).
4. This value can then be copied to another position in the register to be used again later. If not copied, the values will be lost as they will be overwritten by the next operation printed in index 0.
5. Here, we divide index 2 with index 1 to get S/x_n, and copy the value to index 9. Index 9 is the added to index 1 to get $x_n + S/x_n$, and copied to index 8. Divide index 8 by index 11 (constant 2) to get $\frac{x_n + S/x_n}{2}$ and copy to index 7.
6. Now that we have the equation which we need to iterate through a loop (by repeating step 5). With the new approximated value for x (x_{n+1}) we need to decide whether the machine can halt or whether it should continue iterating. We do this by subtracting index 7 from index 1 and square this value to get an absolute value which can be comparable with our stopping criteria constant ($|\varepsilon|$).
7. We create a branch which decides whether to continue iterating or to halt by telling the machine that if $(x_{n+1} - x_n)^2 \leq |\varepsilon|$ 'halt', otherwise continue iterating.
8. If the machine does not halt, S will be divided by the new approximated x found in index 7 and sent back through step 5.

```
def bss(code, outgoing,initValues=[0.0],initNode="1"):
    regs = initValues
```

```
    node = initNode
    counter = 0
    while outgoing.has_key(node) and counter < 50 :
        counter += 1
        print counter, ":", node, code[node]
        nodeType = code[node][0]
        if nodeType == "branch":
            branchType = code[node][1]
            (i, j) = code[node][2:]
            (left, right) = outgoing[node]
            if branchType == "<=" and regs[i] <= regs[j]: node = left
            elif branchType == "==" and regs[i] == regs[j]: node = left
            elif branchType == ">=" and regs[i] >= regs[j]: node = left
            else: node = right
        else:
            (i , j) = code [node][1:]
            if nodeType == "assign": regs[i] = j
            elif nodeType == "copy": regs[j] = regs[i]
            elif nodeType == "add": regs[0] = regs[i] + regs[j]
            elif nodeType == "subtract": regs[0] = regs[i] - regs[j]
            elif nodeType == "multiply": regs[0] = regs[i] * regs[j]
            elif nodeType == "divide": regs[0] = regs[i] / regs[j]
            else: print " unknown"
            node = outgoing [node]
        print regs

code = {}
outgoing={}
#x_(n+1) = (1 / 2) (x_n + S / x_n)

code["1"] = ["divide",2,1]    ; outgoing["1"] = "copDiv"
code["copDiv"] = ["copy",0,9]    ; outgoing["copDiv"] = "add1"
code["add1"] = ["add",1,9]    ; outgoing["add1"] = "copAdd1"
code["copAdd1"] = ["copy",0,8]    ; outgoing["copAdd1"] = "divBy2"
code["divBy2"] = ["divide",8,11]    ; outgoing["divBy2"] = "cop4Crit"
code["divBy2"] = ["divide",8,11]    ; outgoing["divBy2"] = "cop4Crit"

code["cop4Crit"] = ["copy",0,7]    ; outgoing["cop4Crit"] = "crit2Stop1"
code["crit2Stop1"] = ["subtract",7,1]    ; outgoing["crit2Stop1"] = "sqrDiff"
code["sqrDiff"] = ["multiply",0,0]    ; outgoing["sqrDiff"] = "copIt"
```

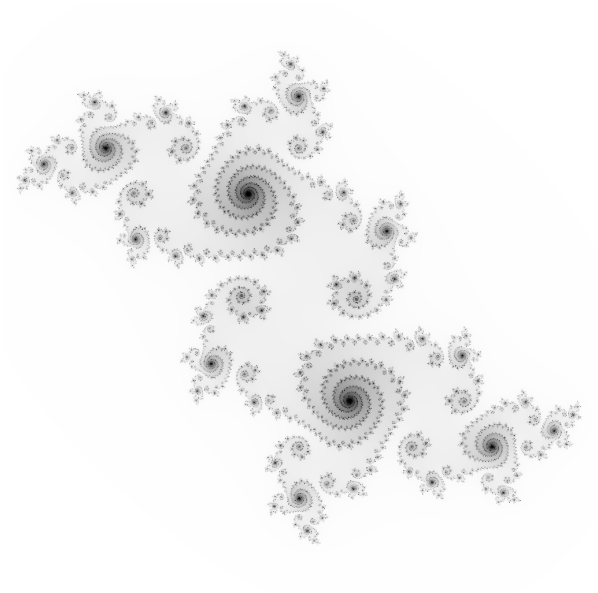
```
code["copIt"] = ["copy",0,6]    ; outgoing["copIt"] = "test"
code["test"] = ["branch","<=",6,12]    ; outgoing["test"] = "halt","2"
code["2"] = ["divide",7,1]    ; outgoing["2"] = "copDiv"
code["halt"] = []

#S = 3.0
#x_1 = 2.0
bss(code,outgoing, [0.0,1.9,3.2,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,2.0,0.0004],"1")
```

Exercise 0.10

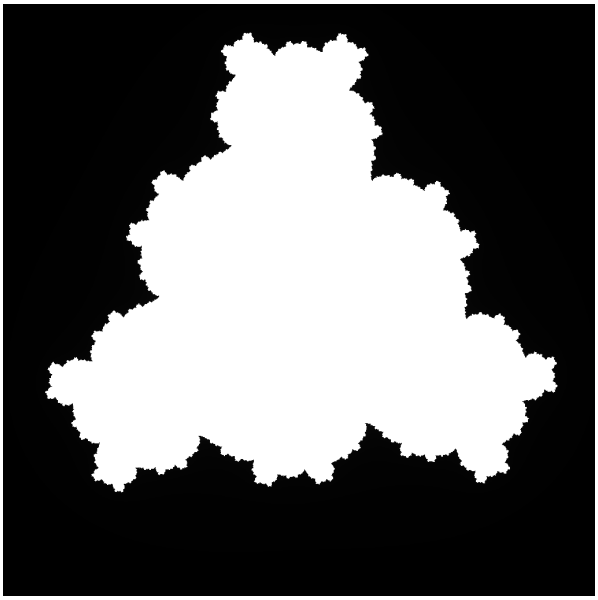
This code prints a grey scale plot of an altered Julia Set equation ($z^3 - 0.065 + 0.66j$ instead of $z^2 - 0.065 + 0.66j$). Plot 1, is the result of this code. The shape of the set changes when $z^2 - 0.065 + 0.66j$ is changed (see Plot 2 and 3).

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

def Julia (f, zmin, zmax, m, n, tmax = 256):
    xs = np.linspace(zmin.real, zmax.real, n)
    ys = np.linspace(zmin.imag, zmax.imag, m)
    X, Y = np.meshgrid(xs, ys)
    Z = X + 1j * Y
    J = np.ones(Z.shape) * tmax
    for t in xrange (tmax):
        mask = np.abs(Z) <= 2.
        Z[ mask] = f(Z[mask])
        J[-mask] -= 1
    return J

zmin = -1.3 - 1j * 1.3
zmax = 1.3 + 1j * 1.3
J = Julia(lambda z: z**3 - 0.065 + 0.66j, zmin, zmax, m=1024, n=1024)
plt.imsave("julia.png", J, cmap=cm.Greys, origin = "lower")
```

Louis Carnec
15204934

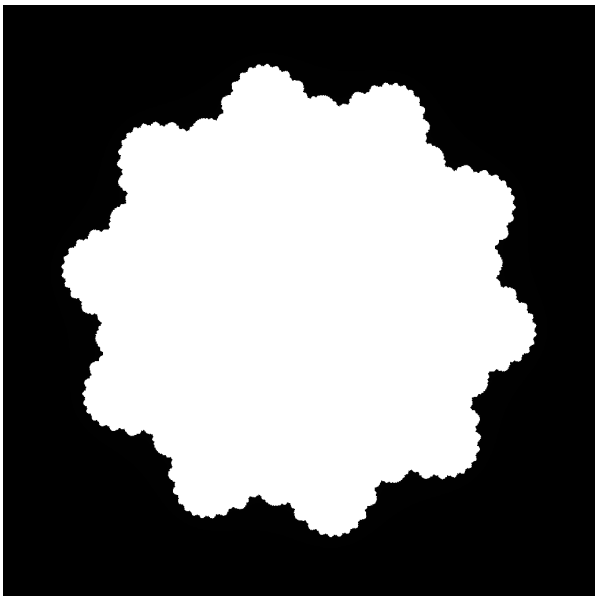


Plot 1: Gray Scale representation of $x^2 - 0.065 + 0.66j$ using colormaps colour $Greys$.



Plot 2: Gray Scale representation of $x^3 - 0.065 + 0.66j$ using colormaps colour $gist\_gray$.

Louis Carnec
15204934



Plot 3: Gray Scale representation of $x^{10} - 0.065 + 0.66j$ using colormaps colour $gist\_gray$.