L1 Practice Exercises
Programming in the λ-calculus

These batch of exercises focus on programming in the pure λ-calculus.

You will use the Python as host language, but the "puzzle" is to code everything just with pure functions, using the basic encodings presented in the M1 Lecture. Write some simple tests to check your code.

We may use the start up code available in the PL Drive.

Learning outcomes:

Refresh basics of functional programming.

Getting familiar with the how the λ-calculus works.

Convince yourself about the Turing-Completeness of the pure pure λ-calculus.

1. This warm up exercise is just to recall the usage of lambda expressions in Python.

a) Write a Python function **iter** that given a function f and a natural number **n** computes the function f^n . Test **iter** by computing

```
v = iter (lambda x:x*2, 10)
print (v (1))
v = iter (lambda x:"+"+x, 10)
print (v ("0"))
```

b) Write a Python function **mapf** that given a list of Python ints $[n_1,n_2,n_3, \ldots]$ and a list of functions $[f_1,f_2,f_3, \ldots]$ from ints to ints computes the list $[f_1(n_2), f_2(n_3), f_3(n_3)]$.

2. Recall the following encoding of booleans in the λ-calculus (in Python)

```
72   true = lambda t: lambda f: t
73
74   false = lambda t: lambda f: f
75
76   ifb = lambda b: lambda t: lambda f: ((b (t) (f)))
```

Write pure λ-calculus functions to represent the boolean operators **not**, **and**, **or** and use the b2str function defined in the lectures to test them is some key cases.

```
bnot =

band =

bor =
```

NB.: in this one and next exercises you may only use pure lambda functions in your functions, no other Python code!

2. Recall the following encoding of pairs in the λ-calculus.

```
278   pair = lambda a: lambda b: lambda f: (f (a) (b))
279
280   fst = lambda p: (p (lambda x: lambda y: x))
281
282   snd = lambda p: (p (lambda x: lambda y: y))
```

Write pure λ-calculus programs to represent the following functions

a) A function **swap** that takes a pair (pair (N) (M)) and returns (pair (M) (N))

b) A function **curry** that takes as argument any function **f** from pairs of values to values and returns the sequentially curried version of **f** so that

for all M,N,V if f (**pair** (M) (N)) = V then (**curry** (f)) (M) (N) = V

2. Recall from the Lectures the encoding of Peano numerals (**pnats**) in the λ-calculus, and the fixed-point combinator **rec** that allows recursive functions to be defined.

You may also use Python conversion functions **p2int** e **int2p** to check your code.

```
104   # Zero := lambda x: lambda f: x
105   # Succ n := lambda x: lambda f: (f (n) )
106
107   zero = lambda x: lambda f: x
108   succ = lambda n: lambda x: lambda f: (f (n) )
109
110   # matchz n { Z -> t | Succ x -> e[x] }
111
112   matchz = lambda n: lambda z: lambda s: \
113                (n (z) (lambda prd: ( s (prd) )))
```

3. Define a function **tri** that takes as argument one **pnat n** and computes the sum all all **pnats** between **0** and **n**, with the following behaviour

```
208   print (p2int (tri (int2p(4))))
209
210   # 10
211
212   print (p2int (tri (int2p(10))))
213
214   # 55
```

4. Define a function **eq** that takes as argument two **pnats** and checks if they are equal (returning a λ-calculus boolean) with the following behaviour

```
230   print( b2str ( eq (int2p(2)) (int2p(2)) ))
231
232   # TRUE
233
234   print( b2str ( eq (int2p(2)) (int2p(10)) ))
235
236   # FALSE
```

5. Define a function **sub** that takes as argument two **pnats m** and **n** and computes m - n, with the following behaviour (notice that m-n = 0 in m<n)

```
167  print( p2int (sub (int2p(30)) (int2p(15))) )
168
169  # 15
170
171  print( p2int (sub (int2p(2)) (int2p(2))) )
172
173  # 0
174
175  print( p2int (sub (int2p(2)) (int2p(5))) )
176
177  # 0
```

6. Define a function **greather_than** that takes as argument two **pnats m** and **n** and checks if m >= n, with the following behaviour

```
239  print( b2str ( greather_than (int2p(2)) (int2p(1)) ))
240
241  # TRUE
242
243  print( b2str ( greather_than (int2p(2)) (int2p(2)) ))
244
245  # FALSE
246
247  print( b2str ( greather_than (int2p(2)) (int2p(3)) ))
248
249  # FALSE
```

7. Define a function **divisible** that takes as argument two **pnats m** and **n** and checks if **m** is divisible by **n**, returning a boolean.

```
278  print(b2str (divisible (int2p(16)) (int2p(5)) ))
279
280  # FALSE
281
282  print(b2str (divisible (int2p(16)) (int2p(4)) ))
283
284  # TRUE
```

8. Define a function **prime** that takes as argument one **pnats m** and checks if **m** is prime, returning a boolean.

You may find it useful to define an aux function **divnone** that takes as argument two **pnats m** and if **n** is prime, and checks that if m is not divisible by any nat between 2 and (**pred** m).

```
296  print(b2str (prime (int2p(67)) ))
297
298  # TRUE
299
300  print(b2str (prime (int2p(69)) ))
301
302  # FALSE
```

8. Recall from the Lectures the encoding of lists in the λ-calculus.

```
342  # NIL := lambda n: lambda c: n
343  # CONS (v, l):= lambda n: lambda c: (c (v) (l))
344
345  NIL = lambda n: lambda c: n
346
347  CONS = lambda v: lambda l: lambda n: lambda c: (c (v) (l))
348
349  # matchl l { NIL -> nc | CONS(v,lr) -> M[v,lr] }
350
351  matchl = lambda l: lambda nc: lambda M: \
352              (l (nc) (lambda v: lambda lr: (M (v) (lr))))
```

Define the functions specified in the next slide.

a) Define a function **list2str** that takes as argument a **list l0** of **pnats** and prints **l0** to a string of the form e.g., [1,2, 6, 20].

b) Define a function **concat** that takes as argument two **lists l0** and **l1** and returns the concatenation of **l0** and **l1**

c) Define a function **find** that takes as argument a **list l0** and a **pnat n** and checks if **n** is in the list (returns a boolean).