CS1010X — Programming Methodology
School of Computing
National University of Singapore

Midterm Test

25 March 2017 **Time allowed:** 1 hour 45 minutes

Student No:	S	O	L	U	T	I	O	N	S

Instructions (please read carefully):

1. Write down your matriculation number on the **question paper**. DO NOT WRITE YOUR NAME ON THE QUESTION SET!
2. This is **an open-sheet test**. You are allowed to bring one A4 sheet of notes (written on both sides).
3. This paper comprises **FIVE (5) questions** and **TWENTY (20) pages**. The time allowed for solving this test is **1 hour 45 minutes**.
4. The maximum score of this test is **100 marks**. The weight of each question is given in square brackets beside the question number.
5. All questions must be answered correctly for the maximum score to be attained.
6. All questions must be answered in the space provided in the answer sheet; no extra sheets will be accepted as answers.
7. The back-sides of the sheets and the pages marked "scratch paper" in the question set may be used as scratch paper.
8. You are allowed to un-staple the sheets while you solve the questions. Please make sure you staple them back in the right order at the end of the test.
9. You are allowed to use pencils, ball-pens or fountain pens, as you like (no red color, please).

GOOD LUCK!

Question	Marks	Remark
Q1		
Q2		
Q3		
Q4		
Q5		
Total		

Question 1: Python Expressions [24 marks]

There are several parts to this problem. Answer each part **independently and separately**. In each part, one or more Python expressions are entered into the interpreter (Python shell). Determine the response printed by the interpreter for the final expression entered and **write the exact output in the answer box**. If the interpreter produces an error message, or enters an infinite loop, explain why. Partial marks may be awarded for workings if the final answer is wrong.

A.

```
a = 2
b = a**2 + a
c = b//a + b%a
if c%b > a:
    print("Higher")
elif c%b < a:
    print("Lower")
else:
    print("Same same!")
```

[4 marks]

```
Higher
```

This is to test that students understand simple `if-else` and also basic operators.

B.

```
def x(y):
    def y(z):
        x = lambda x: x**2
        y = lambda x: x+2
        def z(x):
            return x(5)
        return z(y)
    return y(x)
print(x(lambda x: x+1))
```

[4 marks]

```
7
```

This is to test that students understand function calls and `lambda`.

C.

```
result = 8
for i in range(10,5,-1):
    result = result**2 % i
    if result%2 == 1:
        result = 2*result
print(result)
```

[4 marks]

```
4
```

This is to test that students understand anda can trace through a `for` loop.

D.

```
i, count = 3,0
while i != 1:
    if i%2==1:
        i = 3*i+1
    else:
        i = i//2
    if count > i:
        break
    count += 1
print(count)
```

[4 marks]

```
5
```

This is to test that students understand the `while` loop and `break`.

E.

```
def twice(f):
    return lambda x: f(f(x))
print(twice(twice)(twice(lambda x: x+3))(2))
print(twice(twice)(twice)(lambda x: x+3)(2))
```

[4 marks]

```
26
50
```

This is to test that the students understand the composition of functions.
2 points if one correct in the correct order
1 point if one correct, but in the wrong order (were you guessing?)

Please refer to Appendix Q1E for an in-depth analysis and explanation.

F.

```
def foo(x):
    if x < 4:
        return x
    else:
        return bar(x//2)
def bar(y):
    if y%2:
        return foo(3*y+1)
    else:
        return foo(y//2)
print(foo(10))
```

[4 marks]

```
1
```

This is to test that the students understand mutual recursion and how to trace through mutually recursive code.

Question 2: Continued Fractions [27 marks]

The following is mathematical structural called continued fraction, which is a fraction of the following form:

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\dots + \cfrac{1}{a_n}}}} \tag{1}$$

A. Suppose you are given a tuple of coefficients $(a_0, a_1, \cdots, a_n)$. Implement the function `evaluate_cfraction` that takes as input the tuple of coefficients and returns the value of the continued fraction. [4 marks]

Example execution:

```
>>> evaluate_cfraction((1,))
1

>>> evaluate_cfraction((5,))
5

>>> evaluate_cfraction((3, 3))
3.3333333333333335

>>> evaluate_cfraction((4, 1, 5))
4.833333333333333
```

```
def evaluate_cfraction(coefficients):
    if coefficients == ():
        return 0
    elif len(coefficients) == 1:
        return coefficients[0]
    else:
        return coefficients[0] + 1/evaluate_cfraction(coefficients[1:])
```

Students will not be penalized if they do not deal with empty tuple.

B. What is the order of growth in terms of time and space for the function you wrote in Part (A) in terms of n, the number of coefficients. Explain your answer. [4 marks]

Time: $O(n^2)$, since slicing creates an $n-1$-sized tuple at each level of recursion.

Space: $O(n^2)$, since slicing creates an $n-1$-sized tuple at each level of recursion and they are kept in memory until the evaluation completes.

C. Is the function `evaluate_cfraction` that you wrote in Part (A) recursive or iterative? If it is recursive, implement the iterative version; if it is iterative, implement the recursive version. [5 marks]

```
def evaluate_cfraction(coefficients):
    for i in range(len(coefficients)):
        if i == 0:
            result = coefficients[-1]
        else:
            result = 1/result + coefficients[-(i+1)]
    return result
```

D. What is the order of growth in terms of time and space for the function you wrote in Part (C) in terms of n, the number of coefficients. Explain your answer. [4 marks]

Time: $O(n)$, since we just iterate through each element in the tuple.

Space: $O(1)$, since we only need 2 variables – `i` and `result`.

E. It is great that you can evaluate a continued fraction. Now suppose you are given a rational number $\frac{a}{b}$, where a and b are integers and $b \neq 0$, implement the function `compute_coefficients` that takes in two integers a and b and returns a tuple of the coefficients for the continued fraction of the form:

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ldots + \cfrac{1}{a_n}}}} \tag{2}$$

where a_i is an integer for all i. [5 marks]

Example execution:

```
>>> compute_coefficients(1,1)
(1,)

>>> compute_coefficients(10,2)
(5,)

>>> compute_coefficients(10,3)
(3,3)

>>> compute_coefficients(145,30)
(4,1,5)
```

```
def compute_coefficients(a,b):
    if b == 0:
        return ()
    else:
        return (a//b,) + compute_coefficients(b,a%b)
```

F. Is the function `compute_coefficients` that you wrote in Part (E) recursive or iterative? If it is recursive, implement the iterative version; if it is iterative, implement the recursive version. [5 marks]

```
def compute_coefficients(a,b):
    result = (a//b,)
    while a%b != 0:
        a,b = b,a%b
        result += (a//b,)
    return result
```

Note that the iterative version is actually quite tricky. :-)
-2 points if a and b are not updated simultaneously or with the use of a temp variable, i.e. some students did:

```
        a = b
        b = a%b
```

Question 3: Higher Order Functions [23 marks]

Consider now the following sequence of terms:

$$\begin{aligned} T_0 &= a \\ T_1 &= a+\frac{b}{a} \\ T_2 &= a+\frac{b}{a+\frac{b}{a}} \\ \vdots &= \vdots \\ T_n &= a+\frac{b}{a+\frac{b}{a+\frac{b}{...+\frac{b}{a}}}} \end{aligned}$$

A. **[Warm-up]** Implement the function `compute_nterms(a,b,n)` that computes the *n*th term in the series given *a* and *b*. [3 marks]

```
def compute_nterms(a,b,n):
    if n == 0:
        return a
    else:
        return a + b/compute_nterms(a,b,n-1)
```

The following is the iterative version:

```
def compute_nterms(a,b,n):
    result = a
    for i in range(n):
        result = a+b/result
    return result
```

B. We note the we can define `compute_nterms(a,b,n)` in terms of `fold` (see Appendix) as follows:

```
def compute_nterms(a,b,n):
    return fold(<T1>,<T2>,<T3>)
```

Please provide possible implementations for T1, T2, and T3. [5 marks]

`<T1>:` [2 marks]	`lambda x,y: a+b/y` or `lambda x,y: x+b/y`
`<T2>:` [2 marks]	`lambda x: a`
`<T3>:` [1 marks]	`n`

C. Given the following higher order function `foldl`:

```
def foldl(op,f,n):
    if n == 0 :
        return f(0)
    else :
        return op(foldl(op,f,n-1),f(n))
```

We note the we can define `compute_nterms(a,b,n)` as follows:

```
def compute_nterms(a,b,n):
    return foldl(<T4>,<T5>,<T6>)
```

Please provide possible implementations for T4, T5, and T6. [5 marks]

`<T4>`: [2 marks]	`lambda x,y: a+b/x` or `lambda x,y: y+b/x`
`<T5>`: [2 marks]	`lambda x: a`
`<T6>`: [1 marks]	`n`

D. Given the following higher order function `tail`:

```
def tail(f, a, n):
    if n == 0:
        return a
    else:
        return tail(f, f(n, a), n-1)
```

We note the we can define `compute_nterms(a,b,n)` as follows:

```
def compute_nterms(a,b,n):
    return tail(<T7>,<T8>,<T9>)
```

Please provide possible implementations for T7, T8, and T9. [5 marks]

`<T7>`: [2 marks]	`lambda x,y: a+b/y`
`<T8>`: [2 marks]	`a`
`<T9>`: [1 marks]	`n`

E. Given the following higher order function `head`:

```
def head(f, a, n):
    result = 0
    for i in range(n):
        result = f(result,i,n)
    return result
```

We note the we can define `compute_nterms(a,b,n)` as follows:

```
def compute_nterms(a,b,n):
    return head(<T10>,<T11>,<T12>)
```

Please provide possible implementations for T10, T11, and T12. [5 marks]

`<T10>`: [2 marks]	`lambda x,i,n: a if i == 0 else a+b/x` -1 if didn't deal with `i == 0`
`<T11>`: [2 marks]	Anything will work here.
`<T12>`: [1 marks]	`n+1`

Question 4: Pokemon LOL! [23 marks]

Pokemon are the pocket monsters that populate the world of Pokemon Go! In this problem, you will implement Pokemons. The basic problem is very simple, but in order to fully satisfy the requirements for the problem, you are advised to read through all the subquestions before you start. Your answer for Part(A) will affect your answer to subsequent sub-problems. If you do the wrong thing in Part(A), you might end up not being able to solve later sub-problems.

You are required to implement the following functions in this problem.

1. `make_pokemon` takes in the species of the pokemon, type and combat points and returns a new pokemon with these characteristics.
2. `get_name` returns the species of a pokemon.
3. `get_type` returns the type of a pokemon, i.e. "Water", "Poison", etc.
4. `get_cp` returns the number of combat points for a pokemon.
5. `is_pokemon(obj)` returns `True` if `obj` is a pokemon.
6. `fight(p1, p2)` returns the pokemon, either `p1` or `p2` with the higher number of combat points, or `None` if neither is higher.
7. `evolve` will return a new evolved pokemon. More details given in Part(D) below.
8. `is_same` will compare two pokemons and return `True` if they are the same pokemon. If we compare a pokemon to its evolved form, `is_same` will return `True`.

Example execution:

```
>>> bulbasaur = make_pokemon("Bulbasaur", "Poison", 20)
>>> rattata = make_pokemon("Rattata", "Normal", 10)
>>> squirtle = make_pokemon("Squirtle", "Water",34)

>>> get_name(bulbasaur)
Bulbasaur

>>> get_type(squirtle)
Water

>>> get_cp(rattata)
10

>>> winner = fight(squirtle, bulbasaur)
>>> get_name(winner)
Squirtle

>>> winner = fight(bulbasaur, squirtle)
>>> get_name(winner)
Squirtle

>>> winner = fight(bulbasaur, bulbasaur)
>>> winner
None
```

A. Decide on an implementation for the pokemon object and implement `make_pokemon`, `get_name`, , `get_type` and `get_cp`. Describe how the state is stored in your implementation and explain. [5 marks]

Note: You are limited to using **tuples** for this question, i.e. you cannot use lists and other Python data structures.

Basically, we need to create an object with a unique memory footprint that can be distinguished with `is`.

```
def make_pokemon(name, type, cp):
    return ("pokemon", name, type, cp, None)

def get_name(pokemon):
    return pokemon[1]

def get_type(pokemon):
    return pokemon[2]

def get_cp(pokemon):
    return pokemon[3]
```

Note that we create an extra space in the tuple to track itself. This will be clear in the `evolve` function.

An alternative if you can use list (which you cannot), it will look like:

```
def make_pokemon(name, type, cp):
    return (["pokemon"], name, type, cp)
```

Here, we use `["pokemon"]` to keep track of the pokemon state for Part(D). Unfortunately, `("pokemon",)` doesn't seem to work anymore, but students who use `("pokemon",)` in the same way will not be penalized.

B. Implement the function `is_pokemon(p)` that returns `True` if `p` is a pokemon, or `False` otherwise. [4 marks]

```
def is_pokemon(object):
    return type(object)==tuple and len(object)==5 and object[0]=="pokemon"
```

-2 marks for missing out `type(object) == tuple`.

The alternative looks like this:

```
def is_pokemon(object):
    return type(object)==tuple and len(object)==4 and object[0]==["pokemon"]
```

C. Implement the function `fight(p1, p2)` that returns the pokemon that has the higher number of combat points or `None` if `p1` and `p2` have the same number of points. [4 marks]

```
def fight(p1,p2):
    if get_cp(p1)>get_cp(p2):
        return p1
    elif get_cp(p1)<get_cp(p2):
        return p2
    else:
        return None
```

Pokemons can evolve to become stronger. The function evolve takes a pokemon, a list of possible evolutions and an increase in combat points and returns new evolved pokemon if it is possible. The list of possible evolutions is a tuple of tuples of pokemon species.

Example execution:

```
>>> bulbasaur = make_pokemon("Bulbasaur", "Poison", 20)
>>> rattata = make_pokemon("Rattata", "Normal", 10)
>>> squirtle = make_pokemon("Squirtle", "Water",34)

>>> evolution_list = (("Bulbasaur","Ivysaur","Venusaur"),("Spearow", "Fearow"),\
("Squirtle", "Wartortle","Blastoise"),("Rattata", "Raticate"))

>>> b2 = evolve(bulbasaur,evolution_list,40)
>>> get_name(b2)
Ivysaur

>>> winner = fight(b2, squirtle)
>>> get_name(winner)
Ivysaur
>>> get_cp(winner)
60
```

```
>>> b3 = evolve(b2,evolution_list,30)
>>> get_name(b3)
Venusaur
>>> get_cp(b3)
90

>>> b4 = evolve(b3,evolution_list,40) # cannot evolve!
>>> get_name(b4)
Venusaur
>>> get_cp(b4)
90

>>> b = make_pokemon("Bulbasaur", "Poison", 20)
>>> is_same(bulbasaur,b)
False
>>> is_same(bulbasaur,b2)
True
>>> is_same(bulbasaur,b3)
True
>>> is_same(b2,b3)
True
```

D. Implement the function `evolve` [6 marks]

```
def evolve(pokemon, elist, increase):
    for t in elist:
        name = get_name(pokemon)
        if name in t:
            for i in range(len(t)-1):
                if name == t[i]:
                    if pokemon[4] == None:
                        return (pokemon[0], t[i+1], pokemon[2], \
                                pokemon[3]+increase, pokemon)
                    else:
                        return (pokemon[0], t[i+1], pokemon[2], \
                                pokemon[3]+increase, pokemon[4])
    return pokemon
```

The alternative is much simpler:

```
def evolve(pokemon, elist, increase):
    for t in elist:
        name = get_name(pokemon)
        if name in t:
            for i in range(len(t)-1):
                if name == t[i]:
                    return (pokemon[0], t[i+1], pokemon[2], \
                            pokemon[3]+increase)
    return pokemon
```

E. Implement the function `is_same`. [4 marks]

```
def is_same(p1,p2):
    if not is_pokemon(p1) or not is_pokemon(p2):
        return False
    if p1[4] != None:
        s1 = p1[4]
    else:
        s1 = p1
    if p2[4] != None:
        s2 = p2[4]
    else:
        s2 = p2
    return s1 is s2
```

Again the alternative is much simpler.

```
def is_same(p1,p2):
    if not is_pokemon(p1) or not is_pokemon(p2):
        return False
    return p1[0] is p2[0]
```

2 points for using the `is` operator in a reasonable way.

Question 5: Tell us a Story! [3 marks]

You have been taking CS1010X for about 3 months now. Tell us what you think about the module and also share with us what you think might be an interesting story related to the class.

Student will get points for any reasonably well-articulated answer.

Appendix

The following are some functions that were introduced in class. For your reference, they are reproduced here.

```
def sum(term, a, next, b):
  if (a > b):
    return 0
  else:
    return term(a) + sum(term, next(a), next, b)

def product(term, a, next, b):
  if a > b:
    return 1
  else:
    return term(a) * product(term, next(a), next, b)

def fold(op, f, n):
  if n==0:
    return f(0)
  else:
    return op(f(n), fold(op, f, n-1))

def enumerate_interval(low, high):
    return tuple(range(low,high+1))

def filter(pred, seq):
    if seq == ():
        return ()
    elif pred(seq[0]):
        return (seq[0],) + filter(pred, seq[1:])
    else:
        return filter(pred, seq[1:])

def accumulate(fn, initial, seq):
    if seq == ():
        return initial
    else:
        return fn(seq[0], accumulate(fn, initial, seq[1:]))
```

Scratch Paper

— E N D O F P A P E R —

Appendix Q1E

This is an in-depth look at Q1E (courtesy of a forum post by Qiu Siyu)

1. Background

The following function was provided, which I've renamed and converted to lambda form below.

```
def T(f):
    return lambda x: f(f(x))

[D1] T = lambda f: lambda x: f(f(x))
[D2] A = lambda x: x + 3
```

2. Basics

```
[1.01] T (A)
[1.02] (lambda f: lambda x: f(f(x))) (A)     # from [01]
[1.03] lambda x: A(A(x))     # bring A in
[R1] T(A) = lambda x: A(A(x))
[R2] A2 = T(A) = lambda x: A(A(x))     # just giving it another name
```

It should be clear here that `A2 (or T(A))` applies function `A` twice.

```
[2.01] T(T) (A)
[2.02] (lambda x: T(T(x))) (A)
[2.03] T(T(A))     # bring A in
[2.04] (lambda f: lambda x: f(f(x))) (T(A))     # expand the first T
[2.05] (lambda f: lambda x: f(f(x))) (A2)     # from [R2]
[2.06] lambda x: A2(A2(x))     # bring A2 in
[R3] T(T)(A) = lambda x: A2(A2(x))
[R4] A4 = T(T)(A) = lambda x: A2(A2(x))     # just giving it another name
```

It should be clear here that `A4 (or T(T)(A))` applies function `A2` twice.

Given what we know about `A2` from the previous section, this means `A4` applies `A` 4 times (hence its name).

```
[2.07] T(T(A))
[2.08] (lambda f: lambda x: f(f(x))) (T(A)) # expand the first T.
                                            # Note that this is exactly [2.04]!
[R5] A4 = T(T)(A) = T(T(A)) = lambda x: A2(A2(x))
[R6] A4 = T(T(A)) = T(A2)     # from [R2]
[R7] A8 = T(A4)     # We can generalize this for A{2n} = T(A{n})
```

3. The Question

Using the results from above, this section will work out the 2 statements from Q1E (note that I'm using 0 instead of 2). The idea here is to continuously expand, substitute, and simplify the statements.

```
[3.01] T(T) (T(A)) (0)
[3.02] T(T) (A2) (0)     # from [R2]
[3.03] (lambda x: T(T(x))) (A2) (0)     # expand T(T)
[3.04] # (T(T(A2))) (0)     # bring A2 in. however, this code wouldn't work, so we...
[3.05] (  (lambda f: lambda x: f(f(x))) (T(A2))  ) (0)     # expand the first T
```

```
[3.06] (  (lambda f: lambda x: f(f(x))) (A4)  ) (0)    # from [R6]
[3.07] (lambda x: A4(A4(x))) (0)    # bring A4 in
[3.08] A4(A4(0))    # bring 0 in
[3.09] A4(12)
[3.10] 24


[4.01] T(T) (T) (A) (0)
[4.02] (lambda x: T(T(x))) (T) (A) (0)    # expand T(T)
[4.03] # (T(T(T))) (A) (0)    # T is the argument for the lambda function.
                              # however, this code wouldn't work, so we...
[4.04] (  (lambda f: lambda x: f(f(x))) (T(T))  ) (A) (0)  # expand the first T
[4.05] (  (lambda f: lambda x: f(f(x))) (lambda x: T(T(x)))  ) (A) (0) # expand T(T)
[4.06] (  (lambda f: lambda x: f(f(x))) (T2)  ) (A) (0)  # let T2 = lambda x: T(T(x))
[4.07] (lambda x: T2(T2(x))) (A) (0) # bring T2 in
[4.08] # (T2(T2(A))) (0)    # bring A in. however, this code wouldn't work, so we...
[4.09] (  (lambda x: T(T(x))) (T2(A))  ) (0) # expand the first T2
[4.10] (  (lambda x: T(T(x))) ((lambda x: T(T(x)))(A))  ) (0) # expand the T2
[4.11] (  (lambda x: T(T(x))) (T(T(A)))  ) (0)  # bring A in
[4.12] (  (lambda x: T(T(x))) (A4)  ) (0)  # from [R6]
[4.13] # (T(T(A4))) (0) # bring A4 in. however, this code wouldn't work, so we...
[4.14] (  (lambda f: lambda x: f(f(x))) (T(A4))  ) (0)   # expand the first T
[4.15] (  (lambda f: lambda x: f(f(x))) (A8)  ) (0)   # from [R7]
[4.16] (lambda x: A8(A8(x))) (0)    # bring A8 in
[4.17] A8(A8(0))    # bring 0 in
[4.18] A8(24)
[4.19] 48
```

4. The Pattern

Alternatively... notice from `[R4]` that `T(T)(A)` results in `A` being applied 4 times (i.e. given a value 0, `A(A(A(A(0))))`).

Replacing `A` with `T` results in `T` being applied 4 times (i.e. given a function `A`, `T(T)(A)` `= T(T(T(T(A))))`, resulting in `A` being applied 2**4 = 16 times).

And more generally...

Note: For the comments below, "F applied 6x" denotes this: `F(F(F(F(F(F(x))))))`

```
T = lambda f: lambda x: f(f(x))
S = lambda f: lambda x: f(f(f(x)))
A = lambda x: x + 1    # <-- note that this portion uses +1

T(T(S(A)))(0)    # 12; A applied 3 * 2 * 2 = 12x
T(S(T(A)))(0)    # 12; A applied 2 * 3 * 2 = 12x
S(T(T(A)))(0)    # 12; A applied 2 * 2 * 3 = 12x
S(S(T(A)))(0)    # 18; A applied 2 * 3 * 3 = 18x
T(S(S(A)))(0)    # 18; A applied 3 * 3 * 2 = 18x

(T)(T)(S)(A)(0) # 81; 2nd T applied 2x, thus S applied 2**2 = 4x,
                # thus A applied 3**4 = 81x
(T)(S)(T)(A)(0) # 512; S applied 2x, thus 2nd T applied 3**2 = 9x,
                # thus A applied 2**9 = 512x
(S)(T)(T)(A)(0) # 256; 1st T applied 3x, thus 2nd T applied 2**3 = 8x,
                # thus A applied 2**8 = 256x
(S)(S)(T)(A)(0) # 134217728; 2nd S applied 3x, thus T applied 3**3 = 27x,
                  # thus A applied 2**27 = 134217728x
(T)(S)(S)(A)(0)   # 19683; 1st S applied 2x, thus 2nd S applied 3**2 = 9x,
                  # thus A applied 3**9 = 19683x

T(T(S))(A)(0) # 81; S applied 2 * 2 = 4x, thus A applied 3**4 = 81x
S(T(T))(A)(0) # 64; 2nd T applied 2 * 3 = 6x, thus A applied 2**6 = 64x
S(S(T))(A)(0) # 512; T applied 3 * 3 = 9x, thus A applied 2**9 = 512x
T(T(T))(T)(A)(0) # 65536; T#3 applied 2 * 2 = 4x, thus T#4 applied 2**4 = 16x,
                 # thus A applied 2**16 = 65536x

# 1  2  3 4 5    6
 (T)(T)(T(T(T)))(T(A))(0)
# 1. T#2 applied 2x, thus (T(T(T))) applied 2**2 = 4x
# 2. within the block, T#5 applied 2 * 2 = 4x
# 3. from 1. and 2., T#5 applied 4 * 4 = 16x in, thus (T(A)) applied 2**16 = 65536x
# 4. within the block, A is applied 2x
# 5. from 3. and 4., A is applied 65536 * 2 = 131072x in total

#1 2  3  4 5
 T(T)(T)(T(T))(A)(0)
# 1. T#2 applied 2x, thus T#3 applied 2**2 = 4x, thus T#4-5 applied 2**4 = 16x
# 2. within the block, T#5 applied 2x
# 3. from 1. and 2., T#5 applied 16 * 2 = 32x in total,
#    thus A applied 2**32 = 4294967296x
```