CSC C24H3 S 2017 Final Examination
Duration — 3 hours
Aids allowed: none

Last Name: _____ **First Name:** _____

Student Number: _____ **UTORID:** _____

---

*Do **not** turn this page until you have received the signal to start.*
Good Luck!

This midterm consists of 8 questions on 17 pages (including this one). *When you receive the signal to start, please make sure that your copy is complete.*

- Legibly write your name, UTORID, and student number on this page.
- If you use any space for rough work, indicate clearly what you want marked.
- In all programming questions you may assume all input is valid.
- You do not need to write comments of any kind.

Bonus Marks:	_____/ 10
# 1:	_____/ 18
# 2:	_____/ 10
# 3:	_____/ 9
# 4:	_____/ 16
# 5:	_____/ 20
# 6:	_____/ 15
# 7:	_____/ 20
# 8:	_____/ 10
TOTAL:	_____/118

Question 1. Warming up [18 MARKS]

1. Is the following function tail-recursive? YES NO

```
(define dbl
  (lambda (xs)
    (if (empty? xs) empty
        (cons (* 2 (first xs))
              (dbl (rest xs))))))
```

2. In Scheme, write a linear-time constant-space function `(replace x y xs)` which returns a list just like `xs`, except every occurrence of `x` is replaced by `y`.

3. In Haskell, define an infinite list of primes, called `primes`.

4. Here is my attempt at defining a function `my-or`, which I claim behaves exactly like the expression `(or a b)`:

```
(define my-or
  (lambda (a b)
    (or a b)))
```

   Demonstrate on an example that I am wrong.

5. Now implement this function (called `myOr`, curried, in Haskell-style) in Haskell.

Question 2. Closures [10 MARKS]

What is the output of the following Python program?

```
def meaning_of_life(num_years):
    '''Return the meaning of life as calculated by a supercomputer after
    num_years years.'''

    return '... After %s years ...\n the answer is... %s\n' % (num_years, LIFE)

def report(f, arg):
    '''Return a report produced by f when called with argument arg. '''

    LIFE = "surprise!"
    return 'LIFE is %s\nREPORT: %s' % (LIFE, f(arg))

def report_again(arg):

    LIFE = "Now I'm confused."
    def meaning_of_life(num_years):
        '''Return the meaning of life as calculated by a supercomputer after
        num_years years.'''

        return '... After %s years ...\n the answer is... %s' % (num_years, LIFE)

    return 'LIFE is %s\nREPORT: %s' % (LIFE, meaning_of_life(arg))

if __name__ == '__main__':
    print(meaning_of_life)
    LIFE = 42
    print(report(meaning_of_life, 1000000))
    LIFE = 'Don\'t know.'
    print(report(meaning_of_life, 1000000))
    print(report_again(1000000))
```

Question 3. Type Inference [9 MARKS]

Specify the type of each of the following functions. If you think Haskell's type inference algorithm would fail, write `error`. To get you started and to refresh your memory on Haskell syntax, we've solved the first one for you.

```
f (x, y) = x:y
```

```
f :: (a, [a]) -> [a]
```

```
f1 x y = x y
```

```
f2 (x, y, z) = 42 : map x y
```

```
f3 = foldr (\x y -> x + y + 42.0) 0.0
```

Question 4. Functional features of Python [16 MARKS]

Complete the following Python implementation, so that it produces the specified output.

```
OUTPUT:
-1
  2
    -5
    6
  3
    -7
    8
    9
  -4

1
  2
    5
    6
  3
    7
    8
    9
  4

True
False
1
  2
    5
    6
  3
    7
    8
    9
  4
```

```
class TreeNode:
    def __init__(self, value=None, children=None):
        self.value = value
        if children:
            self.children = list(children)
        else:
            self.children = []

    def __str__(self):
        return str(self.value)

### your functions go here ###

if __name__ == '__main__':

    ROOT = TreeNode(-1,
                    [TreeNode(2, [TreeNode(-5), TreeNode(6)]),
                     TreeNode(3, [TreeNode(-7), TreeNode(8), TreeNode(9)]),
                     TreeNode(-4)])

    print(str_tree(ROOT))
    print(str_tree(tmap(abs, ROOT)))
    print(contains(ROOT, 8, int.__eq__))
    print(contains(ROOT, 42, int.__eq__))
    tmap2(abs, ROOT)
    print(str_tree(ROOT))
```

```
def contains(root, value, equal):
    '''Return if the tree rooted at TreeNode root contains the given
    value. Uses binary function equal to compare values for equality.
    Do not use loops. Use any() and a generator expression. '''

    return
```

```
def str_tree(root, offset=''):
    '''Return a somewhat readable str representation of a tree rooted at
    TreeNode root. See example for format. Do not use loops. Use list comprehension.'''

    return
```

```
def tmap(func, root):
    '''Return a tree that results from applying func to every value stored
    in a tree rooted at TreeNode root.  Do not use loops. Use map and
    a lambda expression.'''

    return
```

```
def tmap2(func, root):
    '''Apply func to every value stored in a tree rooted at TreeNode root.
    Do not use loops. Use list comprehension. '''
```

Question 5. Approaches to Overloading [20 MARKS]

Consider the following two datatypes and complete the implementation below.

```
data Maybe a = Nothing | Something a
data Either a b = This a | That b deriving Show

-- these xs
-- return a list of values from all This's in the list xs.
-- for example, these [This "1", That "2", This "42"]  should return  ["1","42"]
```

```
-- those xs
-- return a list of values from all That's in the list xs
-- for example, those [This 1, That 2, This 42]   should return   [2]
```

What are the types of the above functions?

```
these ::

those ::
```

We now want define a string representation for `Maybe` as follows:

```
> Nothing

> Something 42
42
> Something "Anya"
"Anya"
```

Do so by creating an appropriate `instance`.

Now we want to define equality for the type `Maybe` as follows:
`Nothing` is not equal to anything, and `Something x` is equal to `Something y` if `x` is equal to `y`.
Do so by defining an appropriate `instance`.

Now we define a type class `MyLogic`, which defines a **unary** function `not'`, as well as three **binary curried** functions `and'`, `or'`, and `implies'`. To define an instance of `MyLogic`, it should be sufficient to define **either** `not'` and `and'`, **or** `not'` and `or'`.

We now add the built-in class `Bool` to `MyLogic`, with the "normal" behaviour for the logic functions:

```
> True `and`` False
False
> False `or`` True
True
```

Now we have some fun with logic and put lists in `MyLogic` as well:

```
> not' [True, True, False]
[False,False,True]
> and' [True, True, False] [False,True,False]
[False,True,False]
> implies' [False, True, False] [False,True,False]
[True,True,True]
> and' [[True,False],[True,True]] [[False,True],[True,True]]
[[False,False],[True,True]]
```

Question 6. Algebraic Datatypes [15 MARKS]

Recall the following Haskell datatype from your exercise, and complete the implementation below:

```
data Tree a = Leaf a | Node a (Tree a) (Tree a) deriving Show

many = Node "one" (Node "two" (Leaf "three") (Leaf "four"))
                  (Node "five" (Leaf "six") (Leaf "seven"))

tfold f g (Leaf x) = f x
tfold f g (Node x left right) =  g x (tfold f g left) (tfold f g right)

choose x y z = if z == x then y else z           -- helper for later on

-- fringe t
-- return a list of leaves of t, in left-to-right order
-- for example, fringe many   should return   ["three","four","six","seven"]
```

```
-- replace x y t
-- return a tree just like t, except every occurrence of value x is replaced with y
-- for example, replace "three" "new" many   should return
--    Node "one" (Node "two" (Leaf "new") (Leaf "four"))
--               (Node "five" (Leaf "six") (Leaf "seven"))
```

Now redefine the functions `fringe` and `replace` using a single call to `tfold`, the way you did in your exercise.

Finally, specify the types of these functions:

```
fringe ::

replace ::

tfold ::
```

Question 7. Advanced Inheritance and Operator Overloading. [20 MARKS]

Consider the following starter code and the expected output. Complete the implementation below, so that it produces the expected output. We will consider two `BinaryTree`s equal if they contain the same elements (as compared with `==`), regardless of the shape of the tree. You will need to think carefully about this one!

```
if __name__ == '__main__':

    EMPTY = Empty()
    print('Empty tree:')
    print(EMPTY)
    print('inorder: %s' % EMPTY.inorder())
    print('num_nodes: %s' % EMPTY.num_nodes())
    print()

    TREE = Node(5,
                Node(4, Node(3)),
                Node(9,
                     Node(7, Node(6), Node(8)),
                     Node(10)))
    print('TREE:')
    print(TREE)
    print('inorder: %s' % TREE.inorder())
    print('num_nodes: %s' % TREE.num_nodes())
    print()

    OTHER_TREE = Node(9,
                      Node(4, Node(10), Node(8)),
                      Node(3,
                           Node(5, Node(7), Node(6))))
    print('OTHER_TREE:')
    print(OTHER_TREE)
    print('TREE == OTHER_TREE: %s' % (TREE == OTHER_TREE))

OUTPUT:

Empty tree:
-
inorder: []
num_nodes: 0

TREE:
(((_,3,_),4,_),5,(((_,6,_),7,(_,8,_)),9,(_,10,_)))
inorder: [3, 4, 5, 6, 7, 8, 9, 10]
num_nodes: 8

OTHER_TREE:
(((_,10,_),4,(_,8,_)),9,(((_,7,_),5,(_,6,_)),3,_))
TREE == OTHER_TREE: True
```

```
class BinaryTree(metaclass=ABCMeta):
    '''Abstract Base Class for binary trees.'''

    @abstractmethod
    def num_nodes(self):
        '''Return the number of nodes in this BinaryTree.'''

    @abstractmethod
    def inorder(self):
        '''Return a list of values from this BinaryTree in the in-order
        traversal order.'''

    ### add an appropriate method here ###
```

```
class Empty(BinaryTree):
```

```
class Node(BinaryTree):

    def __init__(self, value, left=None, right=None):
```

Question 8. Parametric Polymorphism [10 MARKS]

Consider the following attempt at implementing the datatype from the previous question in Java.

```
public class Either<X,Y> {

  private X thiss;
  private Y thatt;

  public Either(X thiss) {
    this.thiss = thiss;
  }

  public Either(Y thatt) {
    this.thatt = thatt;
  }

  public X getThis() {
    return thiss;
  }

  public Y getThat() {
    return thatt;
  }
```

TASK: It has a problem. Fix the problem directly in the code above.

TASK: Now implement methods `these` and `those`, that return `List`s, so that the expected output is as follows.

```
  public static void main(String[] args) {
    List<Either<String,Integer>> eithers = new ArrayList<>();      /* OUTPUT:
    eithers.add(new Either<>("csc", 42));                             [csc, c24]
    eithers.add(new Either<>("c24", 24));                             [42, 24]
    System.out.println(Either.these(eithers));                     */
    System.out.println(Either.those(eithers));
  }
```

Bonus. [10 MARKS]

Part (a) [2 MARKS]
Show how to give another name to a Python class.

Part (b) [2 MARKS]
Show how to rename a Python class.

Part (c) [2 MARKS]
Why is the language Python called so?

Part (d) [2 MARKS]
Why is the language Haskell called so?

Part (e) [2 MARKS]
Why is currying called so?

Total Marks = 118