Exam Computer Programming II 2021-12-17

Exam time: 08:00 – 13:00

Practical details and help during the exam:

- The teachers will often be available in https://uu-se.zoom.us/j/61112733698 during the exam You are welcome to spend the exam in that room, as during regular lessons.

  Do not ask questions in the "main room", but go to a breakout room and sign up in the document mentioned below.

- Keep the Google-document
  https://docs.google.com/spreadsheets/d/1zggS0ID0L4Tr7hKbAS4Sb_3D70nedfp9D_tm7fmfZy8
  open during the exam. We will post any late information about, for example, errors in the exam or answers to common questions there.

  Major changes will also be announced in Studium.

  You can also use the Google document to get in contact with the teachers. Fill in name and breakout-room number and we will show up.

- You can also contact the teachers per email (sven-erik.ekstrom@it.uu.se or tom.smedsaas@it.uu.se).

Submission:

- Submission of the exam takes place by uploading the Python files to the same place you downloaded the exam from (like regular assignments).

- The exam has A-tasks and B-tasks. A-tasks must be correctly answered to be regarded as passed. In case of code the code must solve the problem completely. B-tasks can give some credit even if they don't completely solve the problem, however we must be able to run the code!

- Make a zip-file containing all of your files and upload it. If you can't make zip-files you can also upload the files directly (preferably in one drag and drop operation). If you upload the sam file several times it is the last upload that is counted.
  Do not forget to press ''Upload"!

- If there are ant problems with Studium you can e-mail your files to sven-erik.ekstrom@it.uu.se. Remember to include your exam registration code!

- **No submissions after 13:00 will be accepted!**

Rules

- You may not collaborate with anyone else during the exam. You may not copy code or text but you must write your answers yourself.
- You may use the Internet.
- You must write your solutions in the *in designated places* in the files `m1.py`, `m2.py`, `m3.py`, and `m4.py`.
  You must keep the names of the files, classes, methods and functions. Functions and methods must have the parameters as given in the task.
- You may not use other packages than those already imported in the files unless otherwise stated in the task. (The fact that a package is imported *does not* mean that it needs to be used!)
- Before your exam is approved, you may need to explain and justify your answers orally to a teacher.

PLEASE NOTE THAT WE ARE OBLIGATED TO NOTIFY ANY SUSPICION OF ILLEGAL COOPERATION OR COPYING AS A POSSIBLE ATTEMPT TO CHEAT!

Components of the exam:

The exam is divided into four sections, each with tasks corresponding to the four modules. Det finns A-uppgifter och B-uppgifter i alla sektionerna. There are A-tasks and B-tasks in all of the sections.

Grading:

3: At least eight A-tasks passed.

4: At least eight A-tasks passed and either two B-tasks passed or the voluntary assignment.

5: At least eight A-tasks passed and either all B-tasks or three B-tasks and the voluntary task.

Tasks in connection with module 1

The solution to this task must be written in the designated places in the file `m1.py`. The file also contains one `main`-function which tests some parts the code. Feel free to add more tests!

A1: Write the function `number_of_negative(lst)` that returns the number of negative values on all levels in `lst`. The function should handle sublists with *recursion*.

You may assume that all values are numbers.

Exempel:
`number_of_negative(1)` = 0
`number_of_negative(-1)` = 1
`number_of_negative([2, 0])` = 0
`number_of_negative([1,-2,[[-1],[1,-2]]])` = 3
`number_of_negative([1,2,[[1],[-1,-2,-3,-4]]])` = 4

A2: The function below reorders the list that it gets as parameter.

```python
def s_sort(aList):
    for i in range(len(aList)-1):
        m = i
        for j in range(i+1,len(aList)):
            if aList[j] < aList[m]:
                m = j
        aList[i], aList[m] = aList[m], aList[i]
```

How does the time depend on the length n of the list? Answer with a Θ-exprerssion.

You can either motivate the answer with a theoretical reasoning or with a systematic timing.

Hint: If you want to do timing experiments, the included function `random_list` could be useful.

Write the answer in the designated place directly after the function. If you are using timing you have to include your code for that in the `main` function and your timing results in the answer area.

B1: How does the time for the function below depend on the parameter `n`?

```python
def foo(n):
    result = 1
    for i in range(n, n*n*n):
        result += 10*n + (n+2)*n
    return result
```

Answer with a Θ-expression. You have to verify your answer with timing experiments.

Estimate also how long time it would take to calculate `foo(5000)` and `foo(10000)` on your computer.

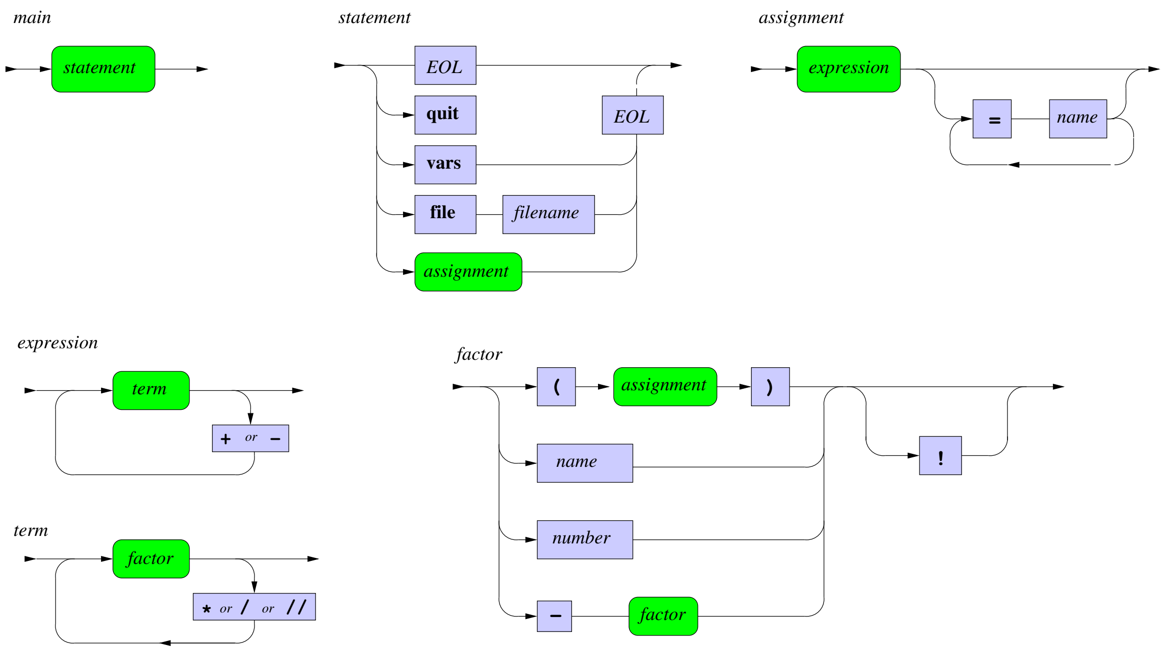
Write your timing code in the `main`-function. and put the output in the designated area at the end of the code.

Note: It must be possible to run the code on other computers!

Tasks in connection with module 2

The downloaded file m2.py implements a calculator like the one in the second assignment. Some constructions are removed and some added. Note that the downloaded file contains the class TokenizeWrapper.

The syntax is defined by the following charts:



The given code contains all functions (green panels) but all of them do not fully function.

The tasks are thus to modify the code so it is compliant with the charts and the description below. You may add more functions if you need.

The command `file` without parameter will cause the calculator to read input from the file `m2_test.txt` which was included in the download.

```
1 - (2 - 4) = x      # 3.0
x*4/2 + x            # 9.0
(1=x) + (2=y)*(3=z)  # 7.0
```

The expected result stands after the comment character #.

After that comes test cases for the tasks which, of cause, don't work until you have solved the tasks

A3: The given code does not handle unary minus. Here follows examples (from the file `m2_test.txt`) of how it *should* work:

```
Input : file
File  : #### A3: Unary minus
File  :
File  : -3                  # -3.0
Result: -3.0
File  : ----3               # 3.0
Result: 3.0
File  : -(3 - 2*3)          # 3.0
Result: 3.0
File  : -3 - 4 + --10 = x   # 3.0
Result: 3.0
File  : -(x+2*-x)           # 3.0
Result: 3.0
```

The syntax is defined in the syntax charts.

A4: Define the operator `//` for *integer division*.

Examples (from `m2_test.txt`):

```
File  : #### A4: The //-operator
File  :
File  : 5//2                # 2.0
Result: 2.0
File  : (5+3)//3            # 2.0
Result: 2.0
File  : (5+3)//3*5//3       # 3.0
Result: 3.0
File  : 5//2.5              # 2.0
Result: 2.0
File  : 5//2.6              # 1.0
Result: 1.0
File  : 5.2//2.6            # 2.0
Result: 2.0
File  : 2///3               # Syntax error
*** Syntax error:  Expected number, word, '-' or '('
Error occurred at '/' just after '//'
File  : 4//(2*3-6)*0.5      # Evaluation error
*** Evaluation error:  Division by zero
```

Division by zero should give an evaluation error.

Hint 1: The Tokenizer returners the sequence `13//3` as three tokens: `'13'`, `'//'` and `'3'`.

Hint 2: The operator should work exactly like the Python `//` operator.

B2: Implement the *factorial operator* `!`. This operator comes *after* its operand. See the syntax charts!

Examples (from `m2_test.txt`):

```
File  : #### B2 Factorial operator
File  :
File  : 0!                    # 1.0
Result: 1
File  : 1!                    # 1.0
Result: 1
File  : 2!                    # 2.0
Result: 2
File  : 3!                    # 6.0
Result: 6
File  : 5!                    # 120.0
Result: 120
File  : 5+ 5! - 10*3!*2       # 5.0
Result: 5.0
File  : 20!                   # 2432902008176640000
Result: 2432902008176640000
File  : 30!/20!               # 109027350432000.0
Result: 109027350432000.0
File  : 51!/50!               # 51.0
Result: 51.0
File  : 10!!                  # Syntax error
*** Syntax error:  Expected end of line or an operator
Error occurred at '!' just after '!'
File  : -3!                   # -6.0
Result: -6
File  : (-3)!                 # Evaluation error
*** Evaluation error:  Argument to fac is -3.0. Must integer >= 0
File  : 1.4!                  # Evaluation error
*** Evaluation error:  Argument to fac is 1.4. Must integer >= 0
```

Tasks related to module 3

In this section you shall work with the file `m3.py` that contains the classes `LinkedList` and `BST` and also the function `random_tree`.

There is also a `main` function with a few tests. You should add more!

The class `LinkedList.py` contains code for handling linked lists of objects. The lists are *not* sorted and the same value can occur several times. Integers are used in the examples but the code should work for all types of objects.

The class `BST` contains code for standard binary search trees.

A5: In the given code for the class `LinkedList` there is a method `append` that adds a new element at the end of the list:

```
    def append(self, x):
        if self.first == None:
            self.first = self.Node(x)
        else:
            node = self.first
            while node.succ:
                node = node.succ
            node.succ = self.Node(x)
```

This method has to start from the first element and iterate throw the whole list to find the place where the new element should be added. It would be better to have another instance variable that points out the last element in the list.

Add such an instance variable in the `__init__`-method in `LinkedList` and write the method `better_append` that uses and maintains this new instance variable and thus get a much better complexity for the operation ($\Theta(1)$ instead of $\Theta(n)$).

A6: What is the (average, not worst case) time complexity for the function below?

```
def random_tree(n):
    t = BST()
    for x in range(n):
        t.insert(random.random())
    return t
```

Answer with an expression of type $\Theta(f(n))$! Motivate!

A7: Write the method `epl` in the class BST that computes and returns the *external path length* (see the course material for definition!)

Example:

```
    t = BST()
    print(f'{t.epl()}␣should␣be␣1')
    t = BST([1])
    print(f'{t.epl()}␣should␣be␣4')
    t = BST([1, 2])
    print(f'{t.epl()}␣should␣be␣10')
    t = BST([1, 2, 3])
    print(f'{t.epl()}␣should␣be␣13')
    t = BST([2, 1, 3])
    print(f'{t.epl()}␣should␣be␣12')
    t = BST([2, 1, 3, 4])
    print(f'{t.epl()}␣should␣be␣17')
```

Output:

```

1 should be 1

4 should be 4

8 should be 8

13 should be 13

12 should be 12

17 should be 17
```

A8: Implement the `==` operator for the class `BST`. The operator should return `True` if the two operands contain exactly the same set of keys (independent of the structure of the trees), else `False`.

Example:

```
t1 = BST([1,2,3])
t2 = BST([2,1,3])
t3 = BST([3,1,2,4])
t4 = BST([1,2,3,4])
print(f'{t1}␣==␣{t2}␣:␣{t1==t2}')
print(f'{t1}␣==␣{t3}␣:␣{t1==t3}')
print(f'{t3}␣==␣{t4}␣:␣{t3==t4}')
```

Output:

```
<1, 2, 3> == <1, 2, 3> : True
<1, 2, 3> == <1, 2, 3, 4> : False
<1, 2, 3, 4> == <1, 2, 3, 4> : True
```

Hint: As seen from the output above, the printouts will be the exactly the same if the trees contains the same keys — a fact that you can use when solving this task. (However, see task B3 below.)

B3: The statement above that the printouts look the same if and only if the trees contain the same keys is not valid if the trees can contain strings.

Example:

```
t5 = BST([1,2])
t6 = BST(['1, 2'])
print(f't5 : {t5}, t5.size(): {t5.size()}')
print(f't6 : {t6}, t6.size(): {t6.size()}')
```

Output:

```
t5 : <1, 2>, t5.size(): 2
t6 : <1, 2>, t6.size(): 1
```

The first tree contains two nodes and the second just one but the printouts are identical.

Write the method `equal(self, other)` that works even if the keys are strings.

Example:

```
t5 = BST([1,2])
t6 = BST(['1, 2'])
print(f't5 : {t5}, t5.size(): {t5.size()}')
print(f't6 : {t6}, t6.size(): {t6.size()}')
print(f't5 == t6 : {t5 == t6}')
print(f't5.equal(t6): {t5.equal(t6)}')
```

Output:

```
t5 : <1, 2>, t5.size(): 2
t6 : <1, 2>, t6.size(): 1
t5 == t6 : True
t5.equal(t6): False
```

The method should have the complexity $\mathcal{O}(n)$ and only make *one* pass over the keys in respective tree. Thus, you may not, for example, make lists of the trees and compare the lists.

Note: If your solution to A8 fulfills these requirements you can just let `equal` use the `==`-operator.

Tasks related to module 4

A9: The function `dice(n,sides)` in `m4.py` simulates `n` throws of a dice with `sides` sides. This function should not be modified.

Modify the function `throw_dice(ns,n_sides)` in `m4.py` so that the following is fulfilled:

- The first argument, `ns`, is a list of the number of throws that should be run in parallel For example, if the argument is `[5, 8, 2]`, then the function `throw_dice(ns,n_sides)` should be run in parallel three processes/threads with 5, 8, and 2 throws (that is, run the function `dice(n,sides)` three times in parallel with `n=5`, `n=8`, and `n=2`).
- The second argument, `n_sides`, should be an integer, that is, all parallel runs should be with the same type of dice (for example, 6 or 20 sides).
- The function should return the quotient between the number of times one got the highest possible value (for example, 6 on a 6 sided dice) and the total number of throws. So, if one have a total number of 100 throws with 6 sided dice, and one gets 14 6:s, then one should return $14/100 = 0.14$.
- You can use any parallelization module that you like, and you should not import any further modules than for parallelization.

A10: When programming you can often get a list of lists, that you want to convert to a list. For example:

Convert `[[1,7,8],[9],[3,3],[1]]` to `[1, 7, 8, 9, 3, 3, 1]`.

Modify `make_list(lst)` in `m4.py` so that the function does this and

- You can assume that `lst` is a list of lists (it is not more than two levels. For example, `[[[1,2], 8], [9]]` is not a valid list).
- The function should be a one row list comprehension.
- You can not import any extra modules.

B4: Among the files that you downloaded for the exam was the file `84-0.txt`, that is cleaned up version of the book “Frankenstenin” (originally from https://www.gutenberg.org/files/84/84-0.txt).

You should modify `count_letters()` in `m4.py` so that is counts how many of each letter (a-z) there is in the text. Big and small letters should be viewed as the same letter, and be counted. You should ignore numbers and other letters.

The function should produce a picture, that is saved to disk, with a bar chart where each bar represents a letter and the height of each bar is defined by the number of times the letter is found in the text `84-0.txt`. The function should also print out the most common letter, and how often it occured,

You can import the modules `string` and `matplotlib` for this task (but no others).

You do not have to upload the picture (it should be generated when grading).