Exam Computer programming II 2024-05-29

Exam time: 08:00 – 13:00

Downloading the exam files

You download the files for the exam from the link in Inspera called Files.

Copy paste contents of downloadable `.txt` files into files named in one of the editors:

`main.txt` –> `main.py`
`m1.txt` –> `m1.py`
`m2.txt` –> `m2.py`
`m2b.txt` –> `m2b.py`
`m2init.txt` –> `m2init.txt` (don't change to `.py`)
`m2binit.txt` –> `m2binit.txt` (don't change to `.py`)
`m2tokenizer.txt` –> `m2tokenizer.py`
`m3.txt` –> `m3.py`
`m4.txt` –> `m4.py`
`m4_data.txt` –> `m4_data.text` (don't change to `.py`)

How to work with the exam

- The exam contains A and B tasks. A tasks must work (submitted programs must be able to run and solve the task) to be approved. B tasks can give "points" even if they don't solve the problems completely.
- Write your solutions *in the places indicated* in the files `m1.py`, `m3.py` and `m4.py`. In `m2.py` and `m2b.py` you have to partly find yourself where the changes and additions should be made.
- In the editors it is always `main.py` that runs. By commenting/uncommenting lines in `main.py` you select which tasks are to be tested.
- It has happened that editors crashes during the exam. Hence, as a precaution take a "backup" of your code by copying your code into Inspera. Do this in as often as you think is necessary. For example, if you work on `m1.py`, copy the contents of that file intp the submission box for `m1.py` in Inspera (ctrl-c to copy and ctrl-v to paste). Then, if the editor crashes you will not lose much time.

Rules

- You must retain names of files, classes, methods, and functions. Functions must be able to be called exactly as stated in the task. If the functions should return something, then it should be according to the specification in the task.
- You may not use packages other than those already imported in the files unless otherwise stated in the task.
- You may write and use help functions.

Submission

To submit you paste (ctrl-v) the content of the whole files `m1.py`, `m2.py`, `m2b.py`, `m3.py`, and `m4.py` in the corresponding boxes in Inspera. A complete submission is hence the contents of 5 files. You should not upload the tokenizer och any `init` or `data` files.

Grade requirements:

3: At least six A tasks passed, of which at least one task passed in each module.

4: At least seven A assignments passed and either two B assignments largely correct.

5: At least seven A assignments passed and four B assignments largely correct.

An approved VA assignment is counted as one B task.

Note that we *can* lower the grade requirements, so it is worth submitting the exam even if you do not strictly meet the requirements stated above.

Tasks in connection with module 1

The solutions to these tasks must be written in designated places in the file `m1.py`.

A1: Write the function `digit_sum(x)` that calculates and returns sum of the digital digits in the non-negative integer `x`.

Example:

```
digit_sum(0)                # should return 0
digit_sum(12)               # should return 3
digit_sum(712)              # should return 10
digit_sum(1021)             # should return 4
```

The task must be solved with recursion and must therefore not contain any iterations nor use any of Python's list or string functions.

The following function is used in tasks A2 and B1:

```
def foo(n):
    if n<3:
        return 1
    else:
        return n + 2*foo(n-1) - foo(n-2) - foo(n-3)
```

A2: The function is only useful for small values on the argument `n`. Write the function `foo_mem` so that it calculates the same result but uses the technique of memoization of calculated results to efficiently be able to calculate, for example, `foo(100)`. The function must therefore still be recursive.

B1: The original version of `foo` was timed using the code below. The printouts are shown on the right. Times are in seconds.

```
for i in (28, 29, 30, 31):
    tstart = time.perf_counter()
    foo(i)
    dt = time.perf_counter()-tstart
    print(f'{i:3d} : {dt:4.2f}')
```

```
28 : 0.96
29 : 1.77
30 : 3.39
31 : 5.99
```

a) Give a Θ-expression for the time complexity indicated by the timing!
b) Estimate how long time the call `foo(100)` would take!
c) Specify a non-trivial Ω expression (ie, a *lower* bound) for the time complexity as the function *guaranteed* has. By "non-trivial" is meant that the expression, although a lower bound, should express the essential behavior of the time complexity!

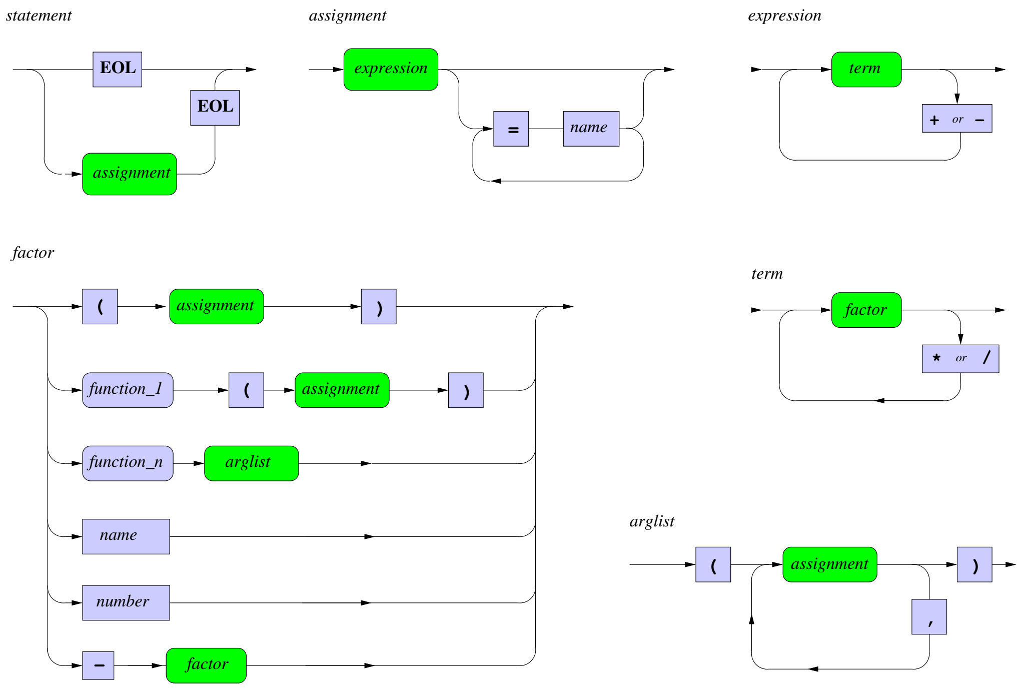
Please note that time measurements in the examination systems are completely unreliable. Use the values given in the task!

All answers should be motivated!

Tasks in connection with module 2

The given file `m2.py` implements a calculator similar to the one in the second assignment.

The syntax of the expressions is described by the following diagrams:



(The given code does not handle functions with multiple parameters.) The tokenizer and an initialization file `m2init.txt` are also included among the downloaded files.

The program starts by reading from the file `m2init.txt` which contains test cases for tasks A3 and A4. If you don't want the program to start reading the test file, you can rename it.

A3: Add || for absolute values. Example:

```

init  : |1-3|                     # Expects 2
Result: 2

init  : ||1-3| - |1-8||           # Expects 5
Result: 5

init  : -|2 + |3-1||              # Expects -4
Result: -4

init  : |1 - 3 +|8|               # Expects SyntaxError
*** Syntax error:  Expected '|'
Error occurred at '# Expects SyntaxError' just after '|'

init  : |1 - 2)|                  # Expects SyntaxError
*** Syntax error:  Expected '|'
Error occurred at ')' just after '2'
```

A4: Modify the code so that a statement can contain several assignments separated by commas.

Example:

```
init  : 1, 2=a, 2+2, 1+1+3, 2+1   # Expects 1, 2, 4, 5, 3
1, 2, 4, 5, 3

init  : ans                       # Expects 3
Result: 3

init  : 3=x, 2=y, x*y=z, z*z      # Expects 3, 2, 6, 36
3, 2, 6, 36

init  : 3, , 4,                   # Expects SyntaxError
*** Syntax error:  Expected number, word or '('
Error occurred at ',' just after ','

init  : 9=x, 3, 4/(2*2-4), 8=y    # Expects EvaluationError
*** Evaluation error: Division by zero

init  : ans, x, y                 # Expects 36, 9, 2
36, 9, 2

init  : (1, 2, 3)                 # Expects SyntaxError
*** Syntax error:  Expected ')'
Error occurred at ',' just after '1'
```

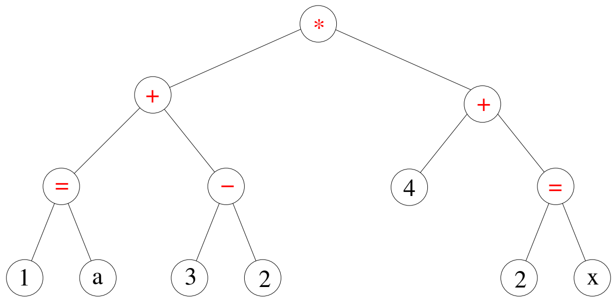
Note that the builtin variable `ans` is set to the value of the last expression on the line and that it is only updated if no syntax or evaluation error has occurred. Also note how results are printed!

B2: This is quite an extensive task that includes elements from both the second and third course modules. It may therefore be wise to save this task until the other tasks are done.

In the second course module, a calculator was written which, using a tokenizer and parser, read and calculated arithmetic expressions. Another way is to let the parser just construct an internal structure of the expression which can then be evaluated. This means that, among other things, you can control the entire expression syntax before any calculations and assignments are performed. It also provides better opportunities for other operations such as differentiation and for defining own functions.

This task involves starting such a program. The program should therefore retrieve input data from the tokenizer as before but, instead of directly evaluating the expressions, build a *binary tree* that represen the expression. The internal nodes represent operations (`+`, `-`, `*`, `/`, `=`) and the leaves constants and variables.

Example: The expression `((1=a) + 3 - 2)*(4 + (2=x))` should be represented by the tree:



When the tree is evaluated, it will result in the value 12 and the variables `a` and `x` get the values 1 and 2 respectively. Note that the parentheses *not* should be included in the tree — it is the structure that determines the order of evaluation.

The file `m2b.py` contains a limited part of the calculator that was included MA2 as well as the skeleton of the classes that will form nodes in the tree. The task is to complete these classes and to modify the calculator so that it, instead of counting, builds up a tree representing the expression. When the tree is ready, it should be *evaluated* and the value printed. It should also be possible to print the expression itself that the tree represents.

The internal nodes should be represented using the `Operator` class and the leaves should represented by the classes `Variable` and `Constant`.

The classes must contain the methods `__init__`, `__str__` and `evaluate`.

The class `Constant` is given and does not need to be changed. The classes `Variable` and `Operator` are partially given but will need modified.

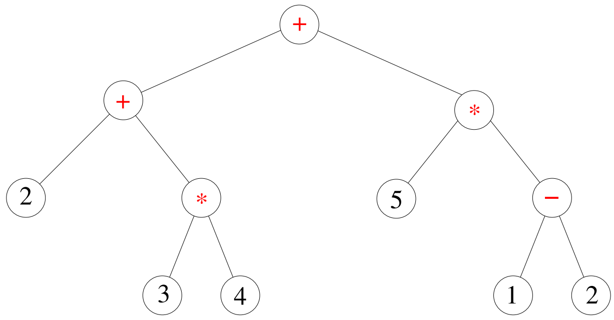
The `Operator` class should represent binary arithmetic operations. These objects constitute the internal nodes of the tree and thus need instance variables to hold its left and right operands.

They must also contain a *name* in the form of a string identifying the operation ie one of the following strings `'+'`, `'-'`, `'*'` or `'/'`.

Subtask 1: Building an expression tree with arithmetic operators

In this, expressions with constants and the binary arithmetic operators, i.e. `+`, `-`, `*` and `/` should be read in and stored in a tree structure. So you should modify the parser functions so that they return node references instead of numerical values. Wait with handling the assignment operation! The `variables` parameter to the parser functions can be removed — it serves no function in parsing.

Example: The expression `2+3*4 + 5*(1-2)` should result in the tree



Printout:

```
Input : 2 + 3*4 + 5*(1-2)
Parsed: 2+3*4+5*1-2
```

Note the lack of parentheses in the printout. This is problem is addressed in subtask 3.

Subtask 2: Evaluate the expressions

To evaluate (calculate) the numerical value of the tree, the nodes must contain the method `evaluate(variables)`. This method is given for the class `Constant` but you must write it for the `Operator` class. Common to the evaluation of the binary operators is that they must first evaluate both of their operands and then perform its own operation on the two evaluated operands.

The function for the operators should be retrieved from the dictionary `OPERATORS` which for each possible operator name stores a lambda expression that defines the operation (cf. handling of function objects in the calculator in MA2!)

In subtask 3 you need also specify a priority for each operator. Therefore store tuples with lambda expression and priority in the dictionary.

The evaluation is started by calling the root node's evaluation function in the `main` function. Make sure the `main` method is updated with this! Using the same example as above, the result is this:

```
Input : 2+3*4+5*(1-2)
Parsed: 2+3*4+5*1-2
Result: 9
```

Subtask 3: Provide the printouts with parentheses

Modify `__str__` in `OPERATOR` so that it puts out parentheses as needed. The needs are controlled by a *priority* associated with each node in the tree. In the given code, the priority of `Constant` and `Variable` is set to 100, which means that you never need to put parentheses around constants and variables. In the dictionary `OPERATORS` you give the second value (after the lambda expression) a priority.

Give `+` and `-` low (and same) priority as well `*` and `/` a slightly higher (and the same) priority. By comparing its own with the children's priority, the `__str__` method in `Operator` can determine whether parentheses need to be placed around each child. Use the priorities instead of looking at the operations — you shouldn't having to change the code if you add new operators!

When that's done, our example expression should produce the following results:

```
Input : 2+3*4+5*(1-2)
Parsed: 2+3*4+5*(1-2)
Result: 9
```

You should not put out parentheses if they are not needed!

Subtask 4: Implement assignments

The assignment operator `=` is indeed a binary operator but it is a bit special because it should only evaluate one of the operands. We have therefore chosen that it should be implemented with a separate class with named `Assignment`.

Write that class and make sure it can be instantiated by the parser. Its evaluation function must, in addition to evaluating the left operand, also put the value in the dictionary `variables`.

When done, our initial example should look like this:

```
Input : ((1=a)+(3-2))*(4+(2=x))
Parsed: ((1=a)+(3-2))*(4+(2=x))
Result: 12

Input : vars
    E      : 2.718281828459045
    PI     : 3.141592653589793
    a      : 1
    ans    : 12
    x      : 2
```

Tasks in connection with module 3

In this section you will work with the file `m3.py` which contains the classes `LinkedList` and `BST`.

The `LinkedList.py` class contains code to manage linked lists of objects. The lists are not sorted.

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

A5: The `push(self, x, index=0)` method in the `LinkedList` class shall insert a new node with the value `x` at the given index. As usual, index 0 means the first place, index 1 the second and so on. The given method is only implemented for the default case, i.e. it inserts in the beginning when no index is given. The task is to complete the method so that it works even when an index is given as a parameter. It should be possible to insert a new element after the last element in the list, but if a larger index than that is specified, a `ValueError` should be created..

Example: The code

```
    lst = LinkedList()
    testset = ((4, 0), (1, 0), (2, 1), (3, 2), (5, 4), (9, 0),
               (7, 7), (6, -1))
    for item in testset:
        try:
            lst.push(item[0], item[1])
            print(f'Pushed {item[0]} at index {item[1]}.' +
                  f' Resulting list: {lst}')
        except ValueError as ve:
            print(f'*** {ve}')
```

should produce the following output

```
Pushed 4 at index 0. Resulting list: (4)
Pushed 1 at index 0. Resulting list: (1, 4)
Pushed 2 at index 1. Resulting list: (1, 2, 4)
Pushed 3 at index 2. Resulting list: (1, 2, 3, 4)
Pushed 5 at index 4. Resulting list: (1, 2, 3, 4, 5)
Pushed 9 at index 0. Resulting list: (9, 1, 2, 3, 4, 5)
*** Index 7 out of range in push
*** Index -1 out of range in push
```

A6: The `remove_largest(self)` method in the `BST` class should remove the node with the largest value from the tree and return the value itself.
Complete the *recursive* helper function `_remove_largest(node)` so that it works in the given `remove_largest`.
Tip: A maximum of 5 lines of code are needed.

Exempel: Koden

```python
inserts = (5, 1, 8, 2, 12, 3, 4, 6)
print(f'Inserted values: {inserts}')
bst = BST(inserts)
print('Removed values:', end=' ')
while not bst.is_empty():
    print(bst.remove_largest(), end =' ')
print(f'\nResulting tree: {bst}')
try:
    bst.remove_largest()
except ValueError as ve:
    print(f'*** {ve}')
```

should produce the following output

```
Inserted values: (5, 1, 8, 2, 12, 3, 4, 6)
Removed values: 12 8 6 5 4 3 2 1
Resulting tree: <>
*** Empty tree in remove_largest
```

B3: Write the method `max_level_sum(self)` in the class `BST` that returns the maximum sum of the keys at the same level.

Example: The code

```
    tree = BST((4, 1))
    print(f'Expects 4, got {tree.max_level_sum()}')
    tree = BST((4, 7))
    print(f'Expects 7, got {tree.max_level_sum()}')
    tree = BST((4, 7, 9))
    print(f'Expects 9, got {tree.max_level_sum()}')
    tree = BST((4, 7, 9, 3))
    print(f'Expects 10, got {tree.max_level_sum()}')
    inserts = (5, 8, 1, 3, 7, 2, 6, 9)
    print(f'\nInserted keys: {inserts}')
    tree = BST(inserts)
    print('Maximal level sum:', tree.max_level_sum())
    tree.insert(12)
    print('Maximal level sum:', tree.max_level_sum(),
          'after inserting 12')
    try:
        print('Empty tree', BST().max_level_sum())
    except ValueError as ve:
        print(f'*** {ve}')
```

should produce the following output

```
Expects 4, got 4
Expects 7, got 7
Expects 9, got 9
Expects 10, got 10

Inserted keys: (5, 8, 1, 3, 7, 2, 6, 9)
Maximal level sum: 19
Maximal level sum: 20 after inserting 12
*** Empty tree in max_level_sum
```

Tasks in connection with module 4

In this section you will work with the file `m4.py` (download name `m4.txt` but name it in the Pythoneditor to `m4.py`). You will also download a file named `m4_data.txt` and it should be called this (not renamed to `m4_data.py`).

You may only import `functools.reduce` and modules/packages for parallelization in this section module 4 (there are also some preimported modules in the file `m4.py`). If you solve the tasks by importing other external modules or packages, the task is automatically failed.

A7: You have a list `v` of the following form

- The list has an even number of elements.
- The list consists of x and y-coordinates for different vectors, for example,

  `v=[1,2.1,4,-2,1,3]`

  represents three vectors

$$
\begin{aligned}
(x_1, y_1) &= (1, 2.1),\\
(x_2, y_2) &= (4, -2),\\
(x_3, y_3) &= (1, 3).
\end{aligned}
$$

Modify the function `lengths(v)` in `m4.py` such that it fulfills the following:

- Create and return a list where every element is the length of every vector (x_k, y_k) in `v`. In the example above we want to return

  `[2.3259406699226015, 4.47213595499958, 3.1622776601683795]`

  since

$$
\begin{aligned}
\sqrt{x_1^2+y_1^2} &= \sqrt{1^2+2.1^2} \approx 2.3259406699226015,\\
\sqrt{x_2^2+y_2^2} &= \sqrt{4^2+(-2)^2} \approx 4.47213595499958,\\
\sqrt{x_3^2+y_3^2} &= \sqrt{1^2+3^2} \approx 3.1622776601683795.
\end{aligned}
$$

- The list `v` can be arbitrarily long, but has alway an even number of elements (you do not need to test this).
- Use at least two higher order functions that has been covered in the course, for example, `zip`, `map`, `reduce`, `list comprehension`, `filter`.
- Write the function on one line.
- Do not use any other modules or packages than `math` and `functools.reduce` (if you want to use reduce). You may not use `numpy`.

A8: In `m4.py` there is a function defined, `simulation(n, scaling)`, that you should **not** modify. The function returns a list of length `n`, with random floats in the interval $[0, \texttt{scaling}]$.

Modify function `run_simulations(n_total, scalings)` in `m4.py` so that it:

- Executes a number of simulations (function calls to `simulation`) in parallel. Import and package/module you want for this.
- `scalings` is a list of different "scalings", the length of this list decied how many processes that should be run in parallel. The variable `n_total` is the total number of samplings that should be used for the simulations. For example, with the call

  ```
  >>> run_simulations(1000, [2.5,3,4])
  ```

  you should in parallel make these three function calls

  ```
  simulation(334,2.5)
  simulation(333,3)
  simulation(333,4)
  ```

  Obeserve that the sum of all the function call's `n` must be `n_total` (above 334+333+333=1000). If you in the example above had `n_total=1001` you should make these function calls,

  ```
  simulation(334,2.5)
  simulation(334,3)
  simulation(333,4)
  ```

  Hence, spread out the simulations as even as possible, and "fill" remainders up to `n_total` in the order of the arguments `scalings`.
- The function should return a list with the same length as `scalings`, where every element is the **sum** of the returned values from each simulation that was run in parallel.
- During the exam, do not test with too many processes (use a maximum of 4) or with large `n_total` (it is enough with 100 or 1000) to avoid overload.

B4: Imagine you are working for a marketing department of a company, and you are tasked to assemble information on products you are selling;your boss wants, for example, to know what age categories are buying the diffferent products to see what demographics they should target with ads.

You get a file for each product (you have for one product, called `m4_data.txt`) where the first row constsis of the model name of the product, standard revenue in SEK (there is a youth and pension discount, se below), and the number of purchases registered in the file. Each rown in the rest of the file contains a number 10-99 that represents the age of a person that bought the product. For example, the beginning of the file `m4_data.txt` looks like this

```
XPZ112 211.90 1000
83
76
73
90
...
```

Model name of the product is `XPZ112`, standard revenue is 211.90 SEK and there are 1000 registered purchases in the file (ages 83, 76, 73, 90, ...). The total length of the file is hence 1001 rows.

You will assemble the data in age categories/bins of size 10, for all ages (that is, 10-19, 20-29, ..., 90-99); nobody under 10 or over 99 years has bought the product.

You will modify the function `get_statistics(filenames, n_processes)` in `m4.py` so that it does the following (read thru everything before coding).

- The argument `filenames` is a list of filenames for different products; for example

  `["m4_data1.txt", "m4_data2.txt"]`

- The function should return a list with the following information for every product (every product information is in itself a list of siz elements)

  1. Model name of the product.
  2. Standard revenue of the product.
  3. A list of number of people in each age category (list with 9 elements).
  4. A list of lists, where each inner list belongs to an age category and contains the ages of all the people in that category; the ages should be sorted from low to high. If more than one person has a specific age in the data, they should all appear in these lists. (list of 9 elements).
  5. Average age of all the people that has bought the product.
  6. The revenue of each age category in SEK (round to closest integer value). Those who are 20 or youer, or 65 and older have a 10% discount.

- The code should be parallelized on two levels:

- Every product should be handled in parallel, each by `n_processes` processes; that is if you have two products then the whole program should use `2*n_processes` processes.
- For each product you should check how large the data file is (on line 1) and then distribute what line numbers should be handled by each process. Henc, you should not read all the data from the file and distribute the date, but you should instead locally read the correct line numbers of the file in each process.

  When every process is done, the data has to be merged so the statistics define above is returned.

- Also, write a function `print_statistics(filenames, n_processes)` that presents the information from `get_statistics(filenames,n_processes)` in a clear way in the terminal (for example, do not print the variable 4 in the list above).

  Example (it doesn't have to be exactly like this):

```
>>> print_statistics(["m4_data.txt"],2)
----------
Model: XPZ112
Standard revenue per sale: 211.9
Average age: 60.547
Total revenue: 201368.57
Bins with number of people and revenues:
[10,19]:    26     4958
[20,29]:    56    11739
[30,39]:    80    16952
[40,49]:   120    25428
[50,59]:   169    35811
[60,69]:   165    33247
[70,79]:   161    30704
[80,89]:   132    25174
[90,99]:    91    17355
----------
```

- Use as much higher order functions as you can
- Write as many helper functions as you want
- Do not import any external modules or packages, except the ones needed for parallelization (and `functools.reduce` if you want to use that)