Compiling Techniques – Final project

Semester: 20L

Author: Wiktor Łazarski 281875

Subject: Macrogenerator without parameters but with nested definitions.

I. General overview and assumptions

An assignment is to design macrogenerator which does not allow to specify any parameters in macrodefinition. However, what is special about this macrogenerator is that it processes macrodefinition without parameters but it can be overcome using nested macrodefinitions which are supported by this macrogenerator.
The available discriminants:

Discriminant	Description
&	Macrodefinition
$	Macrocall

To indicate the end of macrodefinition we reused the sign '&'. Hence, macrodefinition structure presents as follow:

&<name> <body>**&**

Assumption: We must somehow distinguish whether '&' indicates beginning of a new macrodefinition or the end of currently examined one. To somehow overcome this problem I assumed that '&', which is directly followed by name (without any white signs between), is a starting discriminant and '&' , which is directly followed by white sign or macrocall discriminant, is closing discriminant.

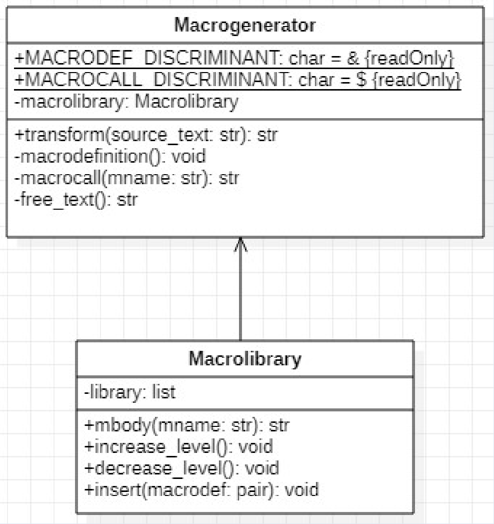
Full project: https://github.com/wiktorlazarski/Macrogenerator

II. Functional requirements

The purpose of macrogenerator is to perform text transformation according to macrodefinitions and macrocalls specified by the user. Because, macrodefinition can be specified by the user inside source text we are dealing with *dynamic* transformation and it is necessary to enhance macrogenerator with a proper macrodefinitions' library data structure to allow hierarchical substitution. More information can be found in *Data structures* section. Examples of usage can be found in *Input/Output* section.

III. Implementation

General architecture

Class UML
 Macrogenerator +MACRODEF_DISCRIMINANT: char = & {readOnly} +MACROCALL_DISCRIMINANT: char = $ {readOnly} -macrolibrary: Macrolibrary +transform(source_text: str): str -macrodefinition(): void -macrocall(mname: str): str -free_text(): str Macrolibrary -library: list +mbody(mname: str): str +increase_level(): void +decrease_level(): void +insert(macrodef: pair): void
Description
Macrogenerator class
MACRODEF_DISCRIMINANT – discriminant indicating macrodefinition.
MACROCALL_DISCRIMINANT – discriminant indicating macrocall.
macrolibrary – Macrolibrary class object used to organize macrodefinitions' library.
transform(source_text: str): str – function transforming *source_text* in a specified way.
macrodefinition(): void – support function to define macrodefinition in macrogenerator.
macrocall(mname: str): str – support function to perform macrocall.
free_text(): str – support function to perform free text reading and writing to output.
Macrolibary class
library – list containing macrodefinitions.
mbody(mname: str): str – function that returns mbody based on mname passed.
increase_level(): void – increases text level (appends new slot for macrodefinitions in library).
decrease_level(): void – decreases text level (remove lastly append slot from library).
insert(macrodef: pair): void – inserts mname and corresponding mbody at current text level.

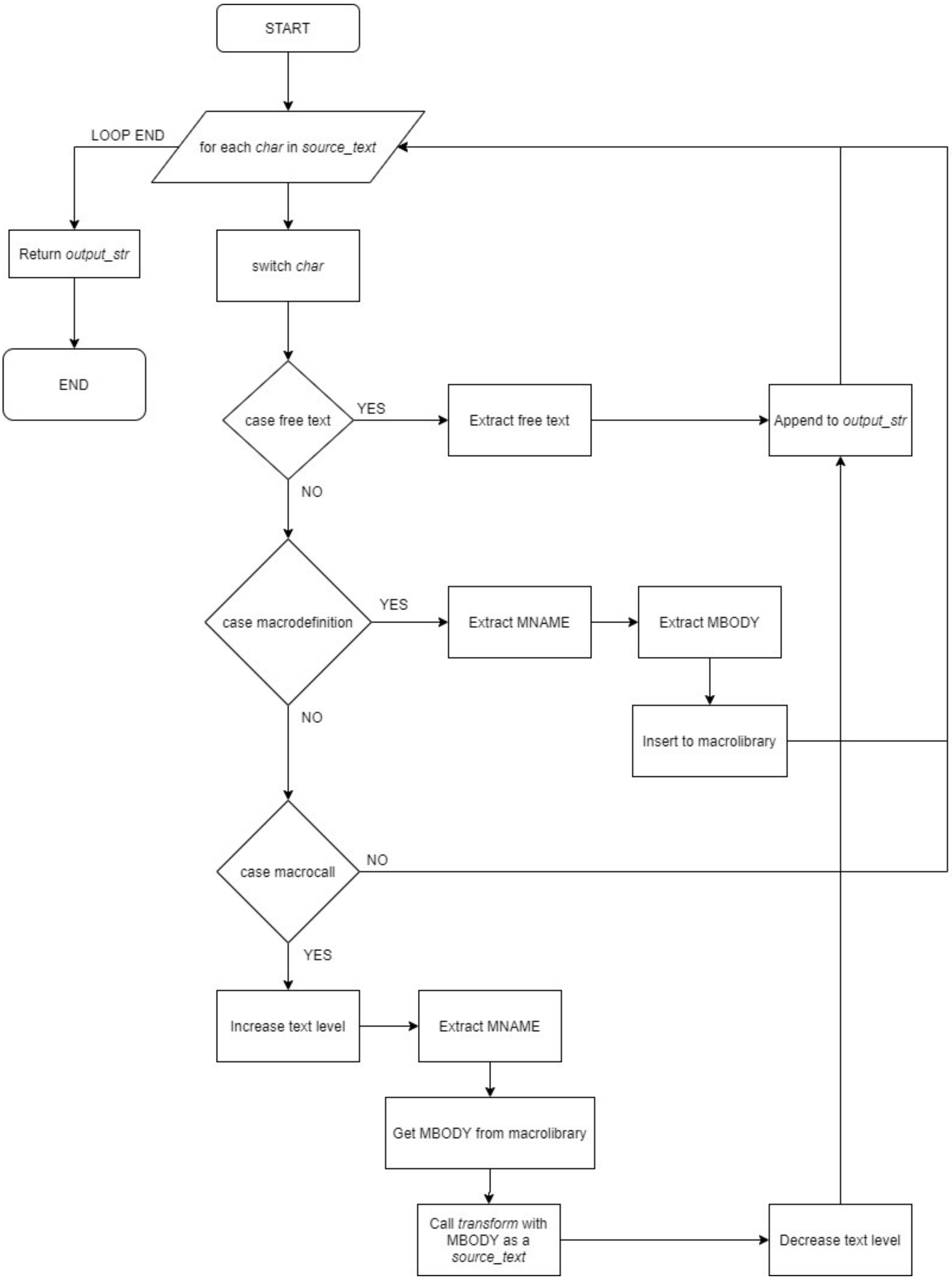
Data structures

Parameter	Description
Macrolibrary	List of macrodefinitions. Due to, the fact that we are going to use hierarchical substitution, indices of a list will indicate current level of text level diagram.
Macrodefinition	Pair of MNAME (macrodefinition name) and MBODY (macrodefinition body).

Remark: At each level of text level diagram more than one macrodefinition may appear. Hence, Macrolibrary must be define as list which allows to store more than one macrodefinition under one index. It may be implemented as a list of hash maps, where each hash map describe macrodefinitions defined at a particular level (list index).

Module descriptions

Main function of a module is a *transform* function. The flowchart below presents data flow through *transform* function.



Flowchart illustrating algorithm of transform function

Remark: It is worth to notice that when macrocall is spotted the function perform recursive substitution. Flowchart does not depict error handling for which I propose exceptions' mechanism.

Input/output description

Macrogenerator takes as an input source text, which may contain macrodefinition and according to them and macrocalls transforms source text and produces output text. Below some examples of transformation performes by macrogenerator.

Example 1: *Basic macrodefinition and macrocall.*

Input	Output
&COMPILE g++**&** **$**COMPILE	*g++*

Where, "*COMPILER*" is a macrodefinition name, "*g++*" - is a macrodefinition body.

Example 2: *Nesting macrodefinition.*

Input	Output
&COMPILE gcc -c **&**NAME main.cpp**&** $NAME**&** **$**COMPILE	*gcc -c main.cpp*

"*COMPILE*" is outer macrodefinition and "*NAME*" is a nested macrodefinition.

Example 3: *Invalid nesting of macrodefinition.*

Input	Output
&COMPILE gcc -c **&**main.cpp **$**COMPILE	*ERROR*

Because of macrogenerator previous assumption, about specifying starting macrodefinition '&' and ending one, *main.cpp* will not be treated as a *free text* but as a new macrodefinition name . Further source text processing will cause an error because there are no macrodefinition closing discriminants ('&').

IV. Functional test cases

Correct source text input test cases

Test case	Description	I/O Example
Basic single macrodefinition	Checks simple single macrodefinition and macrocall of it.	**&**BASIC simple call**&** **$**BASIC Output: “simple call”
Basic multiple macrodefinitions	Checks simple multiple macrodefinitions and macrocalls of them.	**&**BASIC simple **&** **&**NAME call**&** **$**BASIC **$**NAME Output: “simple call”
Macrodefinition nesting	Checks the correctness of macrodefinition nesting.	**&**BASIC simple **&**NAME call**&$**NAME**&** **$**BASIC Output: “simple call”
Proper macrodefinition for a text level	Checks the correctness of choosing the correct macrodefinition for a text level from library.	**&**BASIC simple **&**BASIC call**&$**BASIC **&** **$**BASIC Output: “simple call
Closing macrodefinition discriminant followed by macrocall discriminant	Checks if the macrodefinition closing discriminant is properly validated if next sign is macrocall discriminant.	**&**BASIC simple call**&$**BASIC Output: “simple call”

Incorrect source text input test cases

Test case	Description	I/O Example
Unknown macrodefinition	Checks if macrogenerator returns an error if unspecified macrodefinition is called.	**&**BASIC simple call**&** **$**BASE Output: ERROR: Unknown macrodefinition BASE
Unspecified name of a macrodefinition	Checks if macrodefinition name is declared correctly.	**&** basic**&** **$**BASIC Output: ERROR: Macroname unspecified.
Unspecified body of a macrodefinition	Checks if macrodefinition body is declared correctly.	**&**BASIC **&** **$**BASE Output: ERROR: Macrobody unspecified.
Unspecified closing discriminant of a macrodefinition	Checks if macrodefinition is declared correctly in terms of closing discriminant appears.	**&**BASIC simple call **$**BASE Output: ERROR: Macrodefinition declaration not finished.
Unspecified closing discriminant of a macrodefinition	Checks if macrodefinition is declared correctly in terms of closing discriminant is followed by a white sign.	**&**BASIC simple **&**call **$**BASIC Output: ERROR: Macrodefinition declaration not finished.

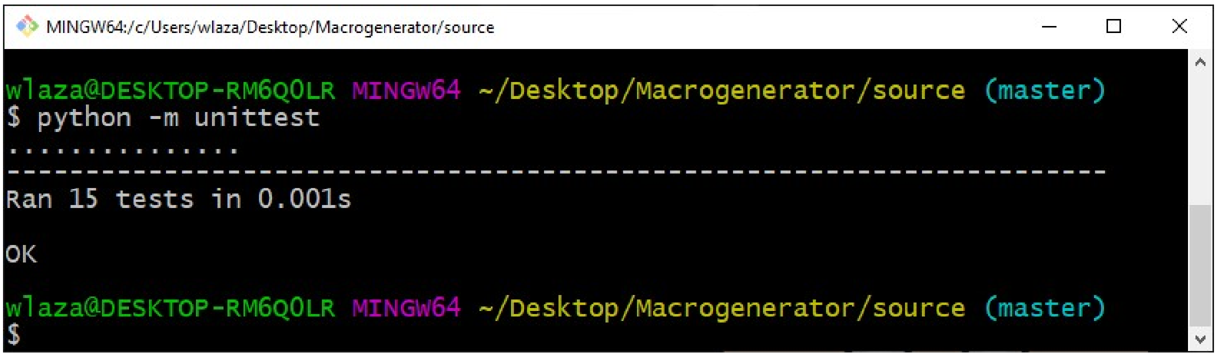
V. Python implementation

Project was implemented using ***Python 3.6*** programming language. Standard Python data structures which were imported to my project are : *list, dict*. To perform unit tests Python *unittest* library was used.

How to run

macrogenerator module is available in *./source* folder. You may run test cases implemented in files: *test_macrolibrary.py* and *test_macrogenerator.py* using following command:

Example 1: Running test cases.

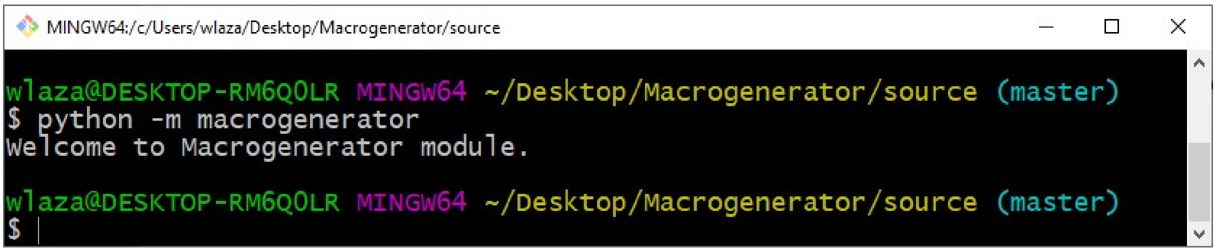


```
wlaza@DESKTOP-RM6Q0LR MINGW64 ~/Desktop/Macrogenerator/source (master)
$ python -m unittest
...............
----------------------------------------------------------------------
Ran 15 tests in 0.001s

OK

wlaza@DESKTOP-RM6Q0LR MINGW64 ~/Desktop/Macrogenerator/source (master)
$
```

Remark: Command must be called inside *./source* folder.

To execute module as a script use the following command, also inside *./source* folder:

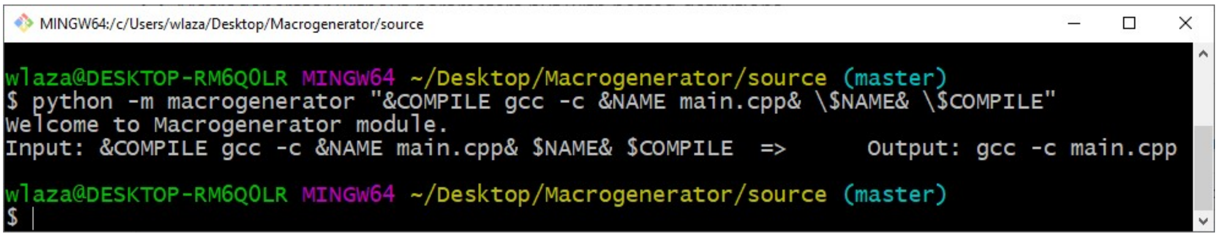
Example 2: Module execution as a script.



```
wlaza@DESKTOP-RM6Q0LR MINGW64 ~/Desktop/Macrogenerator/source (master)
$ python -m macrogenerator
Welcome to Macrogenerator module.

wlaza@DESKTOP-RM6Q0LR MINGW64 ~/Desktop/Macrogenerator/source (master)
$ |
```

Command can be also executed with several command line parameters, making it possible for user to test macrogenerator for different source texts as an input. However, due to, the fact that we are using *'$'* as a macrocall discriminant, we must encounter a problem because terminal will try to read value of a variable followed by *'$'*. Keep it in mind when testing macrogenerator because otherwise you may encounter exception raised by macrogenerator.

Example 3: Transforming source text passed as a command line parameter.



```
wlaza@DESKTOP-RM6Q0LR MINGW64 ~/Desktop/Macrogenerator/source (master)
$ python -m macrogenerator "&COMPILE gcc -c &NAME main.cpp& \$NAME& \$COMPILE"
Welcome to Macrogenerator module.
Input: &COMPILE gcc -c &NAME main.cpp& $NAME& $COMPILE  =>     Output: gcc -c main.cpp

wlaza@DESKTOP-RM6Q0LR MINGW64 ~/Desktop/Macrogenerator/source (master)
$ |
```

Remark: Code available in __*main*__.*py* file.