ECOTE - Final project

Date: 06.05.2023

Semester: 8

Author: Michał Łezka

Subject: Compilation Techniques

I. General overview and assumptions

The goal of this task is to implement a macrogenerator that allows users to define and call macros with key parameters. Macrogenerator will be written in Python and will allow macros to be composed and nested for more advanced functionality. As the input macrogenerator will take a plain text file with macro definitions and calls. Output will be a new text file containing executed macro calls and error log file for all errors that may have happened.

II. Functional requirements

- Macrogenerator can parse macro definitions from the input file to the source code. The algorithm can recognize macro definitions and calls based on the following grammar written in EBNF notation:

macrodef = ‘#MDEF’ name ‘\n’
free_text ‘\n’
‘#MEND’ ‘\n’

macrocall = ‘#MCALL’ name list_of_parameters

name = { letter | number | _ }

list_of_parameters = { name ‘=’ word } | empty

And following not formal definitions:

free_text – any sequence of symbols except for # at the beginning of newline

word – any combination of letters, numbers and symbols until a whitespace or newline

empty – no text

- Hierarchical substitution is allowed. Nested definitions and calls are recognized and stored on the proper text level. If definition of macro 1 is nested inside definition of macro 2, macro 1 can only be called from level of macro 2 higher. If call of macro 2 is finished, all definitions inside it are destroyed and macro 1 can no longer be called.

- Definitions can be overwritten by new definitions with the same name. If the new definition is on higher level than previous definition, it temporarily overwrites it until it gets destroyed.
- Macro calls from the input are executed and printed out in output text file. The algorithm detects and replaces macro calls with its macro code since nested calls are allowed, calls will be called recursively.
- Special signs '#' and '$' can be used as regular characters using escape character '\'.
- Free text will be directly put in the output.
- Errors regarding input will be displayed in console window and saved in error_log.txt file.

III. Implementation

General architecture

The program will be composed of the following modules:

1. Reading source file and storing it in memory.
2. Parsing macro definitions from the source code. This module will identify any macro definitions and extract relevant information such as name, number of parameters, parameters' names, and plain text.
3. Executing macro calls, validating them, and outputting correct macro code.
4. Saves the generated macro code to the output file in the same directory as the input file.

Data structures

To store macro definitions, a dictionary will be used. Name of a macro definition would be used as key and as for the value, it would be a custom Python DataClass with following parameters:
name: str
parameters: set
output: str
macros: {}
called: bool = False
stack_level: int = 0
and will have a single method that will generate proper output from output variable.

Module descriptions

1. Input Reading Module - it is responsible for reading input, splitting it into tokens and storing it in memory. It will provide error-handling for reading files.
2. Macro Definition Parser – it is responsible for detecting and extracting macro definitions from source code to our data structure. It is responsible for checking the correctness of macro definition and handling errors in those definitions.
3. Macro Call Module – its purpose is to locate and process macro calls in the input file. It will identify macro calls, check their correctness, and handle errors in those calls.
4. Write module – writes the result of completed macro calls to the output file and errors to the error file.

Input/output description

As input program expects a .txt file with written program to execute and outputs a processed .txt file together with an error file, e.g.:
Input: some_input.txt
Output: some_input_processed.txt; error_log.txt

IV. Functional test cases

1. Calling basic macro with a parameter.

```
#MDEF Macro_1
Hello $name and welcome!
#MEND
#MCALL Macro_1 name=Mark
```

Output:

```
Hello Mark and welcome!
```

2. Calling basic macro with different order of parameters

```
#MDEF 1
Positive number: $positive
Negative number: $negative
#MEND
#MCALL 1 negative=-5 positive=10
```

Output:

```
Positive number: 10
Negative number: -5
```

3. Nested calls allow to overwrite lower level macros

```
#MDEF 2
Coming from macro 2
#MEND

#MDEF 3
#MDEF 2
Macro 2 was overwritten!
#MEND
#MCALL 2
#MEND

#MDEF 4
#MCALL 2
#MEND

#MCALL 3
#MCALL 4
```

Output:

```
Macro 2 was overwritten!
Coming from macro 2
```

4. Free text remains on output

```
Some free text
```

Outputs:

```
Some free text
```

7. Error – invalid macro name

```
#MDEF .
Hello $name and welcome!
#MEND
```

Outputs:

Error_log.txt:

```
Error: Invalid name ".". It must be made up of only numbers, letters and/or underscores.
Error: Could not generate macro on lines [0:2]
```

Input_processed.txt:

EMPTY

8. Error – invalid macro definition syntax

```
#MDEF Macro_1 Hello $name and welcome!
#MEND
```

Outputs:

Error_log.txt:

```
Error: Incorrect macro declaration: #MDEF Macro_1 Hello $name and welcome!

Error: Could not generate macro on lines [0:1]
```

Input_processed.txt:

EMPTY

9. Error - undefined macro

```
#MCALL macro_5 number=3
```

Outputs:

Error_log.txt:

```
Error: Macro called macro_5 not found.
```

Input_processed.txt:

EMPTY

10. Error – unknown parameter

```
#MDEF Macro_1
Hello $name and welcome!
#MEND
#MCALL Macro_1 name=Mark surname=Doe
```

Outputs:

Error_log.txt:

```
Error: Number of parameters in Macro_1 is 1, was called with 2
```

Input_processed.txt:

EMPTY

11. Error – unknown command after #

```
#WHILE True
Hi!
```

Outputs:

Error_log.txt:

```
Error: Unexpected # in #WHILE
```

Input_processed.txt:

Hi!