Dokumentace úlohy CST: C Stats v Pythonu 3 do IPP 2014/2015
Jméno a příjmení: Jakub Vitásek
Login: xvitas02

1 Assignment

The assignment was to write a script in Python 3, which matches specific data from C language files and returns the number of occurrences of that data. The program uses a number of classes that contain the methods necessary to carry out the main functionality. After these methods are done, the program either outputs the results to the standard output or to a specified file (provided it is writable).

2 Implementation

The program itself can be divided into a number of subroutines. Each one is crucial for the final part to run correctly.

2.1 Getting the user arguments

Python 3 is capable of parsing arguments via a library class `argparse` and its method `ArgumentParser`. All of this is contained within a class `Arguments` which is imported to the main `cst.py` file. Keeping in mind the assignment requires a custom help message, the method is called with the argument `add_help` set to false. At this point, the test for validity of the arguments and their combinations is held.

2.2 Getting the filepaths

Since the user is able to either pass a directory or a file through the `--input` argument, the program has to be able to tell the difference. All of this is contained within a class `Parser` which is imported to the main `cst.py` file. Also, if a directory is passed, the program has to be able to pick usable files, which are only those with the `.c` and `.h` extensions and then add them to the final list of files to parse. It does this using a method called `get_usable_files()`. However, if a single file is passed, the extension is not checked and the file is directly passed into the final list of files.

The same goes for the arguments `-p` and `--nosubdir`. If the argument `-p` is passed, the paths to the parsed files are removed and only filenames are printed. If the argument `--nosubdir` is passed, the program only performs a shallow sweep of the directory (does not go deeper into subdirectories).

2.3 Processing the files

The most crucial part of the program lies within the method `process_file()` in the `Parser` class. It decides which methods to call based on the passed arguments parsed in the `Arguments` class. This method is called iteratively from the main module `cst.py` with each filename. The method always returns the number of occurrences of the particular data. The main module keeps track of the return values and stores the sum in the `total_num` variable, which is printed after all the files have already been printed.

To go over the basic functionality of the parsing methods – which are `get_number_of_occurrences` for `-w`, `get_number_of_operators` for `-o`, `get_number_of_identifiers` for `-i`, `get_number_of_keywords` for `-k` and `get_multiline_comments_number` + `get_inline_comments_number` for `-c` – each one works a little differently. While getting the number of operators, the program removes macros, comments, strings, char literals and pointers using a simple finite-state machine (rather than a regular expression). On the other hand, the program uses a regular expression to get the number of identifiers, keywords and operators. Other operations use the combination of finite-state machine methods like `match_multiline_comments()`, which also accepts an argument remove. If the argument is true, the finite machine removes the matched content and returns the rest of the content. If the argument is false, the method returns the matches.

2.3 Formatting the results

The assignment is very specific considering the output format. The file path strings are meant to be aligned to the left, while the result number is supposed to be right–aligned. Using a very simple cycle, the program finds the maximum length of a file path and makes up for the missing padding by spaces in the other, shorter file paths. To be able to align the numbers to the right, the program also calculates the number of digits in the result number and based on that, it prepends a specific padding to the number. All of that happens in the `format_results()` method of the `Parser` class, which appends the formatted strings to the empty class attribute called `result_strings`. The main module then accesses it, uses the built-in sort method upon the list object to alphabetically sort the output, and then prints it either to the standard output or to a specified output file.