Implementation documentation for the 2st task in IPP 2021/2022
Name and surname: Andrei Shchapaniak
Login: xshcha00

# 1 Interpret.py

## 1.1 Structure of the program

Input arguments are parsed in the function `parse_args()` using `getopt()`. The main arguments are:

- **--help** to print a help message with examples of the program running.
- **--source=** for path to the test source file.
- **--input=** for path to a possible test input.

Additional arguments and their meaning is described in section 1.2.2. After arguments parsing is called function `parse_xml()`, which parses the source file using python module **xml.etree.ElementTree**. Using the `Abstract factory` 1.2.3 design pattern is dictionary `dict_instr` filled by instructions. At the end of the program, when all structures were parsed, `for` loop iterates through instructions.

Different types of **JUMP** instructions, **CALL** and **RETURN** instructions work with variable `_line`, which determines, what instruction will be executed. The result of the program is printed to the standard output.

## 1.2 Implemented extensions

### 1.2.1 STACK

This extension adds to the script new functions which need stack implementation. Examples of instructions: **ADDS** , **NOTS** , **LTS** and others. For this extension, the array `_data_stack` was used which imitates the real stack with functions `push()` and `pop()`.

### 1.2.2 STATI

This extension adds to the script the possibility of gathering different statistics about the source code. To activate it parameter `--stats=filename` must be passed to the script. Instruction **EXIT** is used to finish the program and write statistics to the specified file.

Brief description of the meaning of the parameters. `--insts` counts the number of executed instructions, `--hot` writes **order** attribute of the executed instruction, which was executed the most times and has the value of **order** the smallest, `--vars` writes the maximum number of initialized variables present at one time in all valid frames during the interpretation of program.

Array `_print` is used to save the order of the statistics. Letter `i` means `insts`, `h hot` and `v vars` respectively. To find the value for statistic `hot` is used dictionary `_max_min_op`. This dictionary contains values in the form `"opcode : [call_times , smallest_order]"`. When is instruction **EXIT** executed, function `get_max_min()` finds the desired value.

### 1.2.3 NVI

When implementing this script, the `Abstract factory` design pattern was used. This design pattern was used for the following reasons:

- Classes in the application are not bound into code.

- The program represents the highest level of abstraction. It increases the readability of the code.
- The class of a factory appears only once in the program and creates a complete family of products. It makes it easy to change this factory.

All instructions require the same attributes, therefore class `Instruction` contains **_opcode** - name of the instruction, **_num_of_args** - count of the arguments for this instruction, **_arguments_dict** - dictionary of the arguments in the form `"id_num : {type : value}"` and **instruction_list** - array of the instructions.

For each instruction, there is a class that is inherited from the class `Instruction` and has its own extension in the form of 2 methods:

1. **check_semantic()** provides checking the types of variables, their compatibility, and other semantic controls for the current instruction.
2. **execute()** provides the execution of the current instruction.

The main class is a `Factory` class. There is a **_dict_func** dictionary, which contains a pointer to each class associated with its opcode. There is a method `resolve()`, which performs the main function of the design pattern. When one instruction is passed, `Factory.resolve(opcode, dict_args)` is called. Method knows what it should do with this opcode.

# 2 Test.php

## 2.1 Structure of the program

This program consists of several files, each of which performs its part. **test.php** is the main file which calls functions such as `parseArgs()`, `main()` and `execPython()/Php()/Both()`. The **strings.php** file contains all the strings which are needed to create an HTML file with test results. It builds a big string that prints to the standard output. For these purposes was created a class `Html`, which contains all necessary variables and methods, to provide an easy interface. **errors.php** contains all possible errors that can occur during program running. **arguments.php** provides parsing of arguments using `getopt()` and checks the existence of test files using function `checkFile()`. For input arguments was created a class `Args`, where there are boolean variables responsible for each parameter. There is also a dictionary `arg_path` for arguments that required a path to file. The last file **exec_test.php** contains only class `Test`, where there are methods to execute interpret or parser. There is a method `cleaning()` also. If parameter `--noclean` was not entered, all temporary files will be deleted.

## 2.2 Implemented extensions

### 2.2.1 FILES

This extension adds to the script new ways to select source files for testing.

1. `--testfile=file` parameter is used to specify the list of directories and files with tests instead of loading tests from the current directory.
2. `--match=file` parameter is used to specify the names of the files that match the specified regular expression. It was easy to provide. Before the file will be added to the array of the files with tests, its name is checked using a regular expression.