Documentation of Project Implementation for IPP 2017/2018
Name and surname: **Matúš Liščinský**
Login: **xlisci02**

1 Code analyzer

First task was to create code analyzer for code of imperative language IPPcode18. The script named *parse.php* performs lexical and syntax analysis of code and writes XML representation of the code on standard output. Among other things, script uses **XML Writer**, easy to use PHP extension for generating streams and files containing XML data.

Regular expressions represent important role in analysis and this script demonstrate the awesome power of regular expressions that can save up many many lines of code. With checking if given source code meet the requirements of language IPPcode18 helps multidimensional array.

parse.php: *line 160, 161 resp. 162*

```
/* multidimensional array of instructions and operands */
$instructions = array(
        "MOVE" => array(0 => "var",1 => "symb"),
        ...
```

1.1 Extension STATP

Code analyzer also offers the possibility of collecting statistics about source code in IPPcode18 by additional parameters. Right here is called function **getopt**, which gets these options from the command line argument list. While reading the input file line by line, analyzer recognize if there was a comment or instruction or both and then update prepared stats for printing to *file*.

2 Interpreter of XML representation of code

2.1 XML source file

Interpreter, written in language Python 3, reads XML representation of code in language IPPcode18 and interprets the code. For elegant parsing and working with source XML file interpreter uses **xml.etree.ElementTree** module, which implements a simple and efficient API for parsing and also creating XML data.

2.2 Data structures

In language IPPcode18 there is support for storing data and variables on frames and data stack.
Dictionaries in Python is more than intuitive choice especially for storing information about variables on various frames. ***List*** is replacing a stack, another necessary data structure connected with calling functions (call stack) and naturally data stack.

2.3 About interpretation

Interpreter ensures lexical, syntax analysis and semantic controls of code. Function **sys.exit** is called with explanation of the error in case of failure. Otherwise reads all the instructions and execute them by calling functions utilizing the name of instruction and function **eval**.

2.4 Extension STATI

Script *interpret.py* offers the possibility of collecting statistics about XML representation of code with using additional parameters. The **getopt** module helps script to parse the command line arguments. After each instruction execution, interpreter increment variable used for ***--insts*** choice and call the function to get current number of initialized variables in all the frames (***--vars***).

3 Testing script

3.1 About testing

First action script do is parsing command line arguments, where are informations about location of *parse.php*, *interpret.py*, directory with tests and recursion associated with searching the tests. Recursive searching is covered by **RecursiveDirectoryIterator**. All the tests are saved in one associative array, where key is path to directory with tests. After collecting these data, script is iterating over the associative array, execute tests and generating formatted HTML5 output file.

3.2 Generating HTML5

For creating a HTML5 web page with tests results was used **XML Writer** as in *parse.php*. Output HTML5 file includes information as the success of the tests and directories, returned and expected exit codes, total percentage of tests and so on. Finally, there are few lines of JavaScript code, as way to get to special element of the HTML5 file.