Documentation of Project Implementation for IPP 2017/2018
Name and surname: **Šimon Stupinský**
Login: **xstupi00**

1 Introduction

The task of this project was implementation the two scripts for analysis and interpretation *IPPcode18*, respectively another script for testing this parts. The first pair of the scripts executing the *lexical*, *syntactic* and *semantic* analysis of the relevant form of the code on its input and one of them interpreting the processed code. Gradually, we will introduce the ways of implementation of these scripts, which will be divided into the some parts.

2 Syntactic and Lexical Analysis

The first script, which executing these analyzes is `parse.php`. The input of this script is *IPPcode18* and its task is executing the control of this code. Implementation can be divided into two main parts. The first is representing by the *Finite State Machine (FSM)* and the second using the set of *Regular Expressions*. The *FSM0* is composed of the one general state, which representing the initial and finite state, and other states which represents the individual instructions.

The transitions from this general state depend on the operating code of the currently processed instruction. Subsequently, depending on the state of FSM, comes the combination of both mentioned parts. According to the type of the required argument of the current instruction are executing the relevant actions with use the mentioned set of *Regular Expressions*. This set contains the individual *expressions* to control the allowable constructions of arguments. Specifically, it is the expressions for the relevant types of arguments, thus for *variables*, *label*, *type* and *constants*. In the case that the check runs correctly on all arguments, it will go back to the *general state*. In the case of the errors in some of the arguments, the control will be exit with the relevant error message for users. For the simplicity of this approach and its detailed description, we will not show the example of this FSM.

Syntactic and *lexical* analyzes are executing by the same principle also in `interpret.py`. However, the input in this case is not *IPPcode18* but the *XMLFile*, which contains this processed code. This *XMLFile* is produced by *parse.php*, as its output. Another difference in processing is to check not only the individual arguments of the code but to check the elements of *XMLFile*, and their contents. Not only for these reasons, these scripts contain some additional checks as well as error situations.

3 Semantic Analysis and Interpretation

The actions of semantic control are executing only in the *interpret.py* during the execution of the individual instructions. The implementation of this script it could also mark the approach using the abstraction of *FSM*. At the beginning of the process are executing the control of *Root Element* of *XMLFile*. Subsequently, the all instructions are sorted into the correct order, according to the given number in *order* element of each instruction. The possible redefinition of the labels will be revealed at this stage and the program will be terminated with relevant exit code.

Just as was the case in the *syntactic* and *lexical* analysis, even here have a decisive word the *operating code* of the instructions. At its according will be executing the relevant semantic actions, for example, the types compatibility of arguments. After their success will follow the execution of the given instruction. Such processing will be repeated until the occurrence of failure or executing all the instructions. As one of the advantages or disadvantages of interpreting as such is, that in the case when the error will be somewhere in the later phase of the code, the instruction which are located before this error will be executed.

4 Interesting Facts of Implementation

In this section will be introduced some interesting details about implementation all of the scripts. Arguments of command line are processing with the modules `getopt`[1], which is present in the most of programming languages. To creating *XMLFile* was used the library of PHP `XMLWrite`[2], which meets all the required needs. Contrary, for loading this *XMLFiles* was used module *ElementTree XML API*[3]. For assembling and then testing by using the

[1]https://docs.python.org/3.1/library/getopt.html
[2]http://php.net/manual/en/book.xmlwriter.php
[3]https://docs.python.org/3/library/xml.etree.elementtree.html

Regular expression, was used PHP library function `preg_match`[4], respectively in Python script module *re*[5]. It is worth mentioning mention use the associative array in the *test.php* script, into which are stored test results in its course. After the executed all tests is creating the *HTML* file with the relevant result of these tests, making use of the results stored in the mentioned array. The array also includes counts of successful and unsuccessful tests of both testing modules, with its using, are computed statistics about tests in individual directories or summary statistics. During the development all modules, we created the tests which tested all modules. Module *interpret.py* was examined from the aspect of coverage of the line of code and the results which we achieved are as follows:

```
Coverage for interpret.py: 99%
692 statements 683 run 9 missing
```
[6]

5 Extensions

In the context of this project were implemented some extensions, which are additional to the regular functionality. In the module *parse.php* was implemented the extension, which provides users information about given source file, which contains *IPPcode18.* The first value, which is returned, is the count of lines with instructions, the parameter is known as *LOC — Line of Codes.* These statistics have been recorded during the performance of the individual controls when was detecting relevant admissible instructions. Another value, which is collected, is the count of comments, respectively lines, where the comments are present. This count is incrementing in the case when the line beginning with char, which signs begin of the comment or this char is present at the end of instruction. This fact is authenticated by using the *Regular expression.*

In the second module, *interpret.py* was implemented two extensions. Once serves for collects statistics during the implementation and the second of them extends the set of basic instructions. This statistics for the change provides users information about the count of implemented instruction. After the each successfully executed instruction is counter incrementing. The second parameter of the extension serves to the registration of the maximal count of the defined variables present in all frames during the interpretation.

The last extension which was implemented contains the set of the instructions for work with data stack. These instructions are very similar to standard instructions and it differs only with the count of arguments and its subsequent processing. The logic of executing these instructions was acquired from the mentioned basic sets of instructions and was changed only the methods of obtaining parameters of individual instructions and its stored back. From the data stack is obtained the needed count of instructions, follows is check its type and in the case, when all is right is executing the relevant operation and the result is subsequently stored back to the data stack. In the case, when it need operand from the stack, but the stack is empty, the error message is provided to users and program exit with the relevant exit code.

The script *intepret.py* supports **shorthanded** variant of options from command line. Concrete going about pairs (-s, –source), (-h, –help), (-e, –stats), (-v, –vars), (-i, –insts).

[4]http://php.net/manual/en/function.preg-match.php
[5]https://docs.python.org/3/library/re.html
[6]https://coverage.readthedocs.io/en/coverage-4.5.1/