Documentation of Project Implementation for IPP 2020/2021

Name and surname: Dušan Čičmiš
Login: xcicmi00

Objective

The goal of the project is to create a set of 3 scripts (parse.php, interpret.py, test.php) and interpret the structured and imperative language IPPcode21 using them.

Script parse.php

The job of this script is to check the lexical and syntactic analysis. It loads the program from the standard input and in case of correct code in terms of syntax and lexis, outputs XML representation of the code to standard output. There is a possible extension, which allows users to collect various statistics related to code.

The main part of the script is based on a never-ending loop, which is based on loading a single line of code until there is no line to work with. On every line, the script checks the correct writing of the instruction and in the case of mandatory arguments of the instructions calls function `ArgCheck`, which decides whether the type of a variable and its value is correct. If every line of code is written correctly, it outputs the XML representation of the code, which includes the order and name of instruction (opcode), as well as the value of given arguments.

The extension **STATP** allows users to output desired statistics to a given output file in the case of correct usage of the command line arguments passed to the script (`--stats` followed by wanted statistics). There are seven possible statistics to be gathered: number of labels, jumps, comments, instructions, back jumps, forward jumps and bad jumps. The order of these statistics must be preserved, i.e. they have to be outputted in the same order as they were written to the command line. Function `customGetOpt` checks every argument given and stores them into array `arguments`. For every `--stats` argument, there has to be the same number of valid output files, which is also handled by function `customGetOpt` (stores the name of the files in array `files`). This function then allows function `checkArg` to write desired statistics in correct order, by looping through the array `arguments.`

Script interpret.py

The job of the script is to interpret the code in language IPPcode21. Input of the script is an XML file and in case of the valid XML file, the script interprets the instructions and output is directed to `stdout.`

First of all, the script has to parse command line arguments, what is essential because of determining where to load the XML file from (`--source=XMLfile`). Users can also enter the file, which serves as input for instruction `read`, with `--input=file` switch. When one or both files are missing, the default input method is standard input. Parsing of the command line arguments handles function `parseClArguments.`

In the next stage, the script parses the XML file, with the help of library `xml.etree.ElementTree.` While parsing the XML file, the script appends the instructions with their order, opcode and arguments to the list, using classes `instruction` and `argument` and functions such as `checkArgumentValue` and

`checkInstructionArgument`. The list is later used for interpreting itself. After the list is created and sorted, the script changes the order of the instruction to the form, where first instruction has order = 1 and last one has order = number of instructions. Given the fact that in the XML file, the value of attribute order is not strictly given, this feature comes very useful while interpreting the instructions. After the parsing is completed, the script is ready to interpret the instructions, which i chose to interpret at runtime.

Interpreting is based on while loop, which ends where there is no more instruction to load and execute. Instruction is loaded from the list I mentioned before and is in form: `{'order': 4, 'opcode': 'JUMPIFEQ', 'arguments': ['label', 'end', 'var', 'GF@counter', 'string', 'aaa']}`. In this part, there are 2 carrying elements: class `frame` and `dataStack`. Class `frame` is in charge of handling the job associated with frames such as create stack, push stack to frame stack and so on (there is global and local frame or temporary frames). It provides a solid base for instructions, which initialize, change or simply do anything with variables or constants. Class `dataStack` is used for instructions working with data stack, such as `PUSHS` or `POPS`.