Implementation documentation for the 1. task of IPP subject 2021/2022

Name and surname: Maksim Tikhonov
Login: xtikho00

1 Brief description

parse.php script parses (with the help of supplementary scripts) IPPcode22 code from `stdin` and prints the result XML representation of this code out to `stdout`. All scripts are designed in OOP manner. Remarkable details of implementation just as registered task's extensions are listed and described below.

2 Details

2.1 Parser class

`Parser` class object is instantiated at the beginning of the program execution. Firstly, it parses command line arguments, i.e. `-help`, `-stats=file` and options for stats to be printed out. After that, it starts parsing the input code so, that, first of all, finds mandatory **.IPPcode22** header; then, while parsing individual code lines, trims all comments, skips all empty lines or lines that only contain comments; all "meaningful" lines (ones that contain instructions) are gathered together into array of `Instruction` class objects (`$code`) made by `CodeFactory`.

2.2 Instruction class

Each instance of `Instruction` class represents individual line of code. It contains such attributes as `operation code`, `arrays of arguments` (both for string and `Argument` object representations) and `order`. During the XML document construction each `Instruction` object utilises `makeXMLInstruction()` method, creating DOMElement with corresponding name, operation code, order, arguments and bounding it to parent `program` DOMElement.

2.3 Argument abstract class

Each object of `Argument` subclasses represents single argument of given instruction, it contains only two attributes — `type` and `value` and one method for self-creation. Most lexical and syntax controls are persued upon object creation (decision on what `Argument's` subclass object to create is based on `$instructionSet` associative array from **sets.php**). Argument's `type` attribute definition is based on the string's content before `@`, `value` attribute definition — on the string's content after `@`. I utilised regex patterns for lexical controls of `values` based on different argument types and on individual `type` attributes, e.g. for `Symbol` class of `int` type were used pattern `/^([-+]?[0-9]+|nil)\$/`.

3 Extentions

3.1 STATP

All the output files and corresponding stats options are stored according to command line arguments to associative array `$groups` of shape {filename ⇒ [opt1, opt2, ..., optN]}. Some of the statistics data is obtained during the code analysis (number of comments), others are gained by analysis of already completed list of instructions ($code attribute) by searching for match with jump pattern or with label operation code. All the calculations related to determination of different types of jumps are based on comparisons of `order` attributes of «JUMP *labelname*» and «LABEL *labelname*». After that, all `Stats` class object's attributes that are corresponding to stats' options are printed to corresponding files.

3.2 NVP

As were mentioned before, all parts of code are designed in the OOP way. For the purpose of avoidance of code duplication inheritance were used so, that `Variable`, `Symbol`, `Label`, and `Type` classes are subclasses of abstract class `Argument`. Also I utilised **dependency injection** by creating objects of `Instruction` and `Argument` classes not inside different classes' methods but in **abstract factory** `CodeFactory`. **Singleton** was another OOP pattern (or anti-pattern) I used in `Stats` and `CodeFactory` design; main reason for this was accessibility of objects of both of these classes everywhere in the code.