Implementační dokumentace k 1. úloze do IPP 2022/2023

Jméno a příjmení: Dinara Garipova

Login: xgarip00

Script structure

At the beginning of the script, error return code constants are defined. Definitions are followed by helper functions (writing --help, writing errors, checking program arguments, etc.), followed by functions for checking and writing operands and instructions. Next comes the function for parsing the entire file one line at a time, and finally, the main part of the script. The code uses the **`SimpleXML`** library to create an XML representation of the data.

Script progress

First, a **`SimpleXMLElement`** is created, then the arguments with which the script is run are checked. Initially, all lines from the standard input stdin are stored in the `$lines array[]`, then all comments, blank lines and non-trivial whitespace are removed from there using the `remove_comments_and_whitespace()` function (which replaces the desired comment with an empty line). Immediately after that, using the `separate()` function, the contents of the lines are split into tokens by white characters (the `$line _tokens[]` array), but empty lines (remaining from comments) are also written there. Therefore, from the `$line _tokens[]` array, with a check,if the string is not empty, the tokens are pushed into the `$tokens[]` array, thereby being saved forever without being overwritten.

This is followed by the main loop of the program. It checks if the token of the first non-empty string (`$tokens[0]`) is the header of the program **`".IPPCODE23"`** using the `correct_header()` function, which changes the token to upper case using the `strtoupper()` function before checking, due to the fact that the instruction is not case sensitive and should be displayed in uppercase. If so, the `$is_header_correct` flag is set to true. Otherwise, an error is displayed regarding the loss of the header or its incorrect record. Then, the value of the `$order` variable is incremented and an array of arguments is created using the `array_slice()` function.

After all this, parsing of the file begins using the **`parse()`** function, which works at each stage with the name of the instruction stored in the `$line_tokens[0]` array, the arguments of these instructions, the attribute created in xml, the `$order` variable with the sequence number of the instruction and the number the line on which parsing is in progress, in case an error needs to be output.

The instructions are divided into 8 groups according to the operands they accept with `switch - case` . The first thing in each group is always checked the number of operands, and if it does not match the expected, then an error is displayed. When processing an instruction without operands, the instruction is simply written out in xml using the `print_instruction()` function. For instructions that receive one `<var>` operand, the operand itself is also checked for syntactic and lexical errors using the `check_and_print_arguments()` function, and so on. Finally, from these functions, the functions for processing variables, symbols, labels, and data types are sequentially called (according to the group of instructions).

Here, the correctness of the record is checked directly using a regular expression (using the `preg_match()` function). If there is a match, the necessary information is written into **`XML.`**

If it is an operand of type <symb>, the `check_const()` function is called, if it is a constant, and if it is a variable, then it is simply written using the `print_operand()` function. The `check_const()` function checks the spelling of a constant using a regular expression. To test for "string", the escape sequences are first removed using the `preg_replace()` function. This makes it possible to use a regular expression to check that the text does not contain the '\' character.

Both variables, labels, and constants of the string type will have an explicit replacement for special characters thanks to the `str_replace()` function.

End of script

After the end of the `for` loop, I already have the resulting output in memory. Finally, I write the finished `XML` representation from memory to standard output using the functions of the `SimpleXML` library.