Documentation of Project Implementation for IPP 2018/2019

Name and surname: Michal Pospíšil
Login: xpospi95

Introduction

To improve my OOP skills, the script is object-oriented. Since this is the second time that I'm doing this project, I based my script on the one from the previous year with focus on improving object model, syntax checking and STATP implementation.

Object Model

The `source_code` class handles reading code from the standard input and processes it one line at a time. Its method `process` includes a main loop and behaves as a main function. All member variables are private, `$stats` contains an instance of stats class, `$cur_line` contains unchanged line that is worked on at the moment; `$code_line` contains a line cleaned from an end-of-line character (\r or \r\n, even in files that contain both) and white-space characters surrounding the line; `$cur_line_num` contains line number from original file that is used for error reporting; and `$arg_array` contains a return value of `getopt` function.

Classes `instruction_[0-3]_op` are classes that include instructions loaded from the standard input. Class `instruction_0_op` is a base class – with increasing number of operands these classes are based always on the class with one less operand – leveraging the multi-level inheritance. These classes also include a line number in original file for easier error reporting (`$line_num`).

Next class `rules` is a class that contains methods for checking instruction syntax. For this purpose, two public methods are available – `check_instruct($inst_word)` and `check_vals($instruction).`

For writing XML output, class `xml_out` was created. Methods were created for creating the XML file according to the assignment.

To implement STATP extension, class `stats` is used. Stats are always recorded, even if the --stats argument is missing, but they are not printed into a file in that case.

Implementation Details

Since the source language – IPPcode19 – is quite simple, a decision to not use a finite state machine was made. Instead, the program starts to read lines from the standard input with a loop (`source_code::process`). With some exceptions, all valid constructions in IPPcode19 language use the same structure – (optional white-space character/s) instruction word (white-space character/s) optional argument 1 (white-space character/s) optional argument 2 (white-space character/s) optional argument 3 (optional white-space character/s) end of line character – everything else is marked as an error. The few exceptions are the blank lines, comments and a header.

Cleaning Lines from Blank Lines and Comments (`source_code::clean_line`)

Because the comments are not passed to output, they are removed in the first step. This is achieved with function `preg_split` with pattern '/#/'. Only the first string of the array that this function returns is kept for next processing. In this step, blank lines can also be detected – they present as one string arrays in the return value, that are either empty or contain only white-space characters. These are skipped immediately with `continue` (next line is processed). There is also a special case, when there is more than one string in the returned array, these are comments or code lines with comments – in this case, occurrence of a comment is marked in the `stats` class.

Header

The header is treated as a special case of an instruction. In the beginning of the main loop, there is an if clause that skips blank lines (or those with white-space characters) and comments (as mentioned before, they present as blank lines). When the header is found (with method `source_code::check_header`), this clause is skipped in other iterations.

When the lines contain only the code, they are saved into `$code_line` member variable. Then the line is split into lexemes with function `preg_split`. When the syntax of the first lexeme (should be an instruction name) is checked and expected argument types are returned by `rules::check_instruct`, number of lexemes is checked, appropriate instruction class is created and `argX_type` members (if needed) are filled with expected operand types. Then the operands are checked with `rules::check_instruct` call. As a last step, the instruction is written into XML with call to `xml_out::new_instruction`.

STATP Extension

Extension is implemented in `stats` class. To record number of code lines, comments, labels and jumps, member variables were used. Public methods were defined to increment any of the statistics. Then there is a special method `write_file($args)` where `$args` is an array returned by getopt. If –stats argument wasn't defined then the method does nothing, otherwise it prints the statistics into a file with a foreach loop that iterates through the `$args` array to print values in the same order as they appeared in the command.