Documentation of Project Implementation for IPP task 1
2018/2019
Name and surname: Samuel Stuchly
Login: xstuch06

The goal of this project was to take a code in `IPPcode19`, parse it and accordingly generate xml representation of given code. We had to perform syntactic and lexical analysis to determine if the code was to be made xml representation from, or whether to terminate the program accordingly to the nature of occurred error. I decided to approach the lexical analysis through the use of regular expressions. I made eight variables containing regular expressions (from now regexes) needed for parsing the code. They can be found on the beginning of the source code in marked section List of regular expressions. First function I made was to parse the very first line of given IPPcode19 code, where I got rid of whitespaces with `trim` function, then with the use of `comment` regex filtered out comment from the line and finally I split the line with the use of `preg_split` function and `spaces` regex. From this function I got an array with the separate words from the line. Then I used `preg_match` function and regex `ippcode` to find out if the header is according to rules. If not, error 21 was raised. For the rest of the file I basically used the same principle. I trim the line, cut out the comment, split the line with whitespaces and then use `preg_match` function to determine whether the first word in created array is in fact a Instruction keyword. From instruction set I determined eight groups of instruction. Each group contained instructions followed by particular number and order of operands. Each group had its own regex that was again used with the `preg_match` function. I used a long if elseif else structure to compare the first word. There was instructions with zero, one, two or three operands, so I made functions for each number of operands, except for zero because that could be checked very simply. Each of these functions moved through the array of words form the line and compared each element with regex for the type it was supposed to be. This basically handled the syntactic analysis of the code and therefore could progress to generating the xml code. For generating xml I used XMLWriter library. I wrote two functions, for generating instruction and for generating argument. I used these to functions inside the one, two, three operands functions mentioned earlier to generate xml. Into `generateArgument` function had to be input the type of operand to be written as attribute of element for argument. Type `symb` was problematic because it could consist of either variable or constant. Therefore I had created a function to recognise the type from `symb` and return it. Another function was needed later for making text to be printed between the argument tags. This text had to be adjusted towards the requirements in some cases with use of `strtoupper` function or first part of the string cut out with `explode` function. Other issue that came up was use of certain characters such as `(&,<,>,',“)` in the xml code since these characters could disturb the format of xml. The solution was to replace them with xml entities for these characters. Function used for the replacement was `str_replace`. In case of these characters appearing in the string argument I had to make another function because of use of escape sequences. This function used `preg_replace` functions and regexes for escape sequences representing the characters. With this was my program complete and functional. I tested multiple correct and incorrect scenarios for the functionality and after only a few small adjustments I managed to refine the script.

I also decided to implement extension STATP. Extension required four counters to be made to record number of instructions, comments, labels and jumps in the IPPcode19 code. Counter for instructions was already implemented because each instruction element in xml had to have its number as `order` attribute. Order started from one so for use in STATP it just needed to be decremented. Jumps and comments counters were easy to implement. Every time function one of instructions containing jumps was read the counter was incremented. The assignment was not particularly straightforward in which instructions contain jumps. After looking at forum I decided to count `call` and `return` instructions as jump as well. For comment counter I just checked every line when parsed for hashtag and if it since it could appear in code only as sign of comment. The labels counter seemed easy at first, I implemented it just as incremented counter every time label instruction was read. Although later I read on the forum the question about recording labels read or unique labels read, and thanks to it I noticed I have before misread the assignment. So I created and array for labels and every time label instruction was read I checked if operand of the instruction was already in the `labelsArray` and only if it was not I have incremented the counter. For printing out the correct counters to stats file in correct order I have implemented an array of possible parameters and a loop that would iterate through the array and compare it to each parameter the script was run with. I used the `getopt` function to load a file for stats to be written in. Then thanks to a few of forum threads I set up some errors connected with opening a file and parameter `stats` not being given or being empty.