Documentation of the 2. Project Implementation for IPP 2020/2021
Name and surname: David Mihola
Login: xmihol00

Implementation details

This document describes the logical structure of the interpet and test scripts source code and the approach used to accomplish the assigned task. The more technical details (f.e. implemented functions) are described directly in the source code of each script.

Interpret Script

This script implemenst an interpret of the IPPcode21 languge represented in a XML format.

Command Line Arguments

Command line arguments are parsed by the use of the `getopt` library. If the input of the interpreted program is specified from a file, then the content of that file is read in to an input buffer before interpretation, which is then used instead of reading from `STDIN`. On the other hand, when the source code is specified from a file, the default option of source code input (`STDIN`) is replaced by the specified filename without any further effect on the following implementation, as this filename is passed to the XML parsing function instead.

XML Source Code Parsing

The XML representation of the source code is parsed by the use of the `xml.etree` library. The XML input is parsed to an internal representation of the interpreted program. The internal representation of the program is a list of instructions. If the order of obtained instructions is not continuous, the missing indices of the list are filled with a `NOP` instruction. An instruction consists of an opcode and arguments. If the arguments are constant values, they are converted from their string representation to their corresponding type when being parsed. Furthermore, two dictionaries are created, one with kyes being the order of instructions and the values being the interpreted program labels and the second one with keys as the labels and values as the order of the corresponding instruction. These dictionaries are created when jump or label instructions are parsed and are used to control the program flow, as they work as a transitive relation.

Interpretation

The interpretation is performed by calling functions performing the task of a given instruction in an order obtained from the list specified above, the list is indexed by an instruction pointer. The instruction pointer is increased by one with each executed instruction, but can be changed by instructions performing jumps. These instructions change the instruction pointer according to the values stored in the two dictionaries described above. A function to be called to perform an instruction is determined again by a dictionary. This dictionary consist of instruction names as keys and values being the corresponding functions.

Extensions

I chose to implement the FLOAT and STACK extensions as neither of them required any major change in the logic of the implementation. The FLOAT extension needed support for an additional type during the parsing and additional type checks when performing mathematical and logical operations. The STACK extension required an implementation of additional functions performing the instructions with `S` suffix. Both extensions are compatible with each other, as a result additional `INT2FLOATS` and `FLOAT2INTS` instructions were added, which are not required by the assignment.

Test Script

This script is used for testing the IPPcode21 parser from the first part of the assignment and also the interpret.

Command Line Arguments

The command line arguments are parsed by the use of the `getopt` function. The default values such as the tested directory, the name of a parser script ect. are then replaced by the obtained values, if there are any. Invalid combination of arguments such as `int-only` and `parse-only` are detected and in this case the test ends with an error.

Test File Structure

Firstly, all test files are collected to a multidimensional tree-like array structure. The leaf arrays are filled with test files of the same name. In case of recursive search for the test files, the tree also contains internal arrays representing directories of the file structure. The root array is the searched directory. Secondly, this obtained tree-like structure is analyzed for missing files, which are generated. If there is also the source file missing, then this set of files is not used. As the tree-like structure is analyzed, it is symplified just to a single test sample name for each group of files. Both of the above described algorithms are called recursively, if needed.

Testing

Testing is also performed recursively, if specified in the command line arguments, based on the collected tree-like structure of the test samples. This tree-like structure is then filled with `true` or `false` values based on the success of the test. The tests itself are performed in two steps. The first step is when it is possible to send data to the tested program through a pipe. Then the `proc_open()` function is used to accomplish the task. Secondly the `exec()` function is used to compare the output of the tested program with a reference output. This time the obtained output data of the tested program are saved to a temporary file, which is then used by the comparison programs.

Results Page

The test results are generated in a HTML list tree structure, which corresponds to the tested file structure. To make the results more appealing to a human eye, the succeeding tests are highlighted with a capital green passed tag and in the opposite the failing tests are tagged with a capital red failed. Furthemore, there is a `JavaScript` function added, which allows the user to hide and display certain tested folders. Finally all tested folders have a summary and additionally there is also one main summary of the whole test set at the end of the page.

Test Samples

To ensure that both the parser script and the interpret script are working correctly, I have created a test directory called `tests`. This directory contains test files solely for the parser, solely for the interpret and also for both of the scripts to run in a pipeline.