Implementační dokumentace k 2. úloze do IPP 2018/2019

Jméno a příjmení: Gabriel Quirschfeld
Login: xquirs00

1. Interpret

1.1. Assignment

The goal of this assignment was to create the interpreter script for the IPPcode19 language. The input of the script is the output of the parser created in the previous assignment, so the XML document with the structure of the programming language. As output the interpret prints the results of the instructions to the standard output. The script was supposed to be created in Python3.6.

1.2. Solution

1.2.1. Input arguments

I used the `getopt()` built-in function to parse the input arguments. Everything is handled in the `check_argv()` function, the exceptions included. Only one of input or source arguments can be left unused and vars or insts arguments cannot be used without the stats argument.

1.2.2. XML parsing

The XML is parsed using the ElementTree library. It is read from either the stdin or a user defined file and checked in stages. First I checked the header for the language=IPPcode19 attribute. The structure of all the instruction elements is checked in a simple cycle. The arg elements are filtered through another cycle and ordered based on the number in the name of the element. I have created an object for both the instruction and argument elements for storing them.

1.2.3. Lexical & syntactic analysis

Both the lexical and syntactic analysis are run at the same time inside the `parse_instruction(int mode)` function. Firstly, the number of arguments is checked. The types and the data itself is checked by using regular expressions. It is very similar to the implementation of the analysis in the parser script.

1.2.4. Interpretation

The interpretation of the instructions is done in two runs. Both runs are done in the `parse_instrution(int mode)` function (mode 0 is for lexical and syntactic analysis, 1 is for interpreting). In the first run all the labels are found and stored for the second run. The instructions are interpreted sequentially by their operation code.
I have created two dictionaries (for GF and TF) and two lists (for LF and the data stack). The local frame and temporary frame are left uninitialized.
The variables are dynamically checked. Every instruction with arguments calls the `check_symb()` or the `check_dest()` function that finds out whether the variable exists in the correct frame and if the frame is even initialized.
I have also created the `get_type()` and `set_type()` functions to dynamically check the type of the variable or symbol if the instruction has any restrictions in terms of typing.
Jumps are the only instructions that have a return value. This value is the operation code of the instruction the program is supposed to jump to.

1.2.5. Addons

I have implemented the FLOAT, STACK and STATI addons.
For the FLOAT functionality I mainly added a new allowed data type. I have used the `fromhex()` and `hex()` retyping accessible inside of the python language. Also a new DIV instruction with floating point division is defined.

For the STACK functionality I added new instructions for manipulating the data stack. I have used a list for creating the data stack because the `append()` and `pop()` built-in list functions are very similar to the push and pop instructions of stacks.
For the STATI functionality I added a new input argument for the stati output file. Every time the interpret succeeds in interpreting an instruction, the number of insts is increased. The max number of vars is checked while interpreting every instruction as well.

2. Test script

2.1. Assignment

The goal of this assignment was to create a script for automatizing the testing of the previously created parser and interpreter scripts. The input of the tester are .src (input for the parser, or in case of int-only for the interpreter), .in (input for the read function), .rc (expected return code) and the .out (expected output of the interpreter) files. The script was supposed to be created in PHP7.3.

2.2. Solution

2.2.1. Input arguments

The parsing of the arguments is done through the `getopts()` built-in functions. The exceptions and storage of input data is done in the `parse_arguments()` function. In case any of the arguments is missing the default values are set.

2.2.2. Testing the parser

The parser uses the .src file as input. First I have checked the specified directory (and its subdirectories in case of recursivness) for all the .src files. Any of the needed files are always loaded through the `get_files()` function which uses recursion. If the .rc file doesn’t exist it is created with the default value. The parser is run through the `exec()` built-in function for each of the .src files in a loop. The stderr is redirected to /dev/null and the output is redirected to an .xml file for later use. The return code is compared to the corresponding .rc file and the table cell is generated.

2.2.3. Testing the interpreter

The interpreter uses the output of the parser, the .xml file, as the input. In case of the int-only input argument, the .src file is used instead. The directory is checkes for .src, .in, .rc and .out files (if they were not loaded already) as in the case of the parser. The .xml files are also searched for in general use. If the .in or the .out files do not exist, they are created as empty files. The interpreter is run through the `exec()` function for every input file in the directory. The output is redirected to a .txt file which is then compared to the .out file.

2.2.4. HTML generation

The HTML output is generated separately for every call of the parser and interpreter. I have used the DOMDocument third party structure. The HTML file has a general header describing the purpose of the test script and a table containing the results. The table contains the name of the .src file, expected and actual return codes for both the parser and the interpret and the expected and actual output. If any of the actual values are correct, they are highlighted with a green background, if not a red background is used. If the int-only or the parse-only arguments are present, only the necessary cells are created for the table.

2.2.5. Addons

I have not implemented the addon for the tester.