Implementační dokumentace k 2. úloze do IPP 2022/20223

Jméno a příjmení: Dinara Garipova

Login: xgarip00

Program structure

At the beginning of the source code, there are 4 classes (`Args`, used to parse the program's input arguments, `ProgramXMLReade`r to check the input xml file, `Instructions` to parse instructions and their arguments, and `Interpret` is the main working part of the program for interpreting the code). At the end of the source file is the `main function`, which incrementally executes the program, creating the desired program classes.

Script progress

The program starts with a main function that creates an instance of the `Args` program class.

This is where `bool` base flags are created to store the flags in the input arguments, if any. Next comes the verification of the input arguments, their number and interaction with each other.

The paths to the input file are also checked immediately (their correctness). And there is an implementation of issuing `--help` if necessary.

An instance of the `ProgramXMLReader` class is then created. In addition, the `parse_file()` method parses the source file with an XML representation. For this I used the `xml.etree.ElementTree` library. Next, it checks the correctness of the XML notation, and also checks if a certain root tag has only allowed attributes using the methods
`def_check_structure_of_xml_tree(self),def_check_instruction_order.`

Next, the `Instructions` class is created for the main function, where the "skeleton" of the input file is passed as an argument.

Dictionaries for instructions, orders and labels are created here. The instructions themselves are checked (does the program know the instructions used), the number of their arguments and their types.

And also, each instruction is written to the self.`instr_dict dictionary` with the keywords and arguments "opcode", "args", "type".

And it is also important to note that already at this stage a dictionary with labels `self.label_dict` is created and filled, which is necessary for jumps and function calls.

Next, the most important class in the program is created, which does all the main work - `Interpret`, which is passed as arguments a dictionary of instructions, a dictionary with labels and an inputfile (necessary for example for the `READ` instruction).

The main function of this class is to interpret the code, but there are a few helper functions here:

`rewrite_string` - Helps to convert a string in string format into the proper form for writing out.

`getfromvar` - This function was created to cut down on repetitive pieces of code in a file. It can be used to access the type and value of an argument if it is represented as a variable that is stored in memory.

And finally, the most basic function is `interpret.`

The `scope` counter is required to work with local variables and their visibility in certain parts of the input code

First, a dictionary with the names of instructions is created or overwritten in the loop, as well as dictionaries with arguments and their types, respectively.

Many instructions are interpreted in a roughly similar scenario, so let's look at one of them to understand the implementation of the interpretation in more detail.

Consider, for example, working with the `IDIV` instruction:

Initially, `help_stack = [], typ = {}, value = {}` help dictionaries and lists were created because of a small error caused by working with variables stored in memory. The fact is that the self.getfromvar function, in order to make it easier to work with the type and value of a given argument, overwrites its type and value with those that are in the memory of this variable in order to return the type and value back and these auxiliary lists are used.

After working with memory, the arguments and their types are checked for correctness (their existence, type consistency, and correctness of types for a particular instruction).

Next comes splitting the value of the first argument using the `split('@')` function. This is necessary to understand what kind of memory we are working with, and then to which variable we should write the result of the instruction.

Be sure to check for the existence of the variable to be written to.

During the execution of the instruction itself, the type is checked, and division by 0 is also prevented (in this case).

After writing the result to a variable, the modified type and values return to their original values.

After traversing the entire list of instructions, the program terminates.