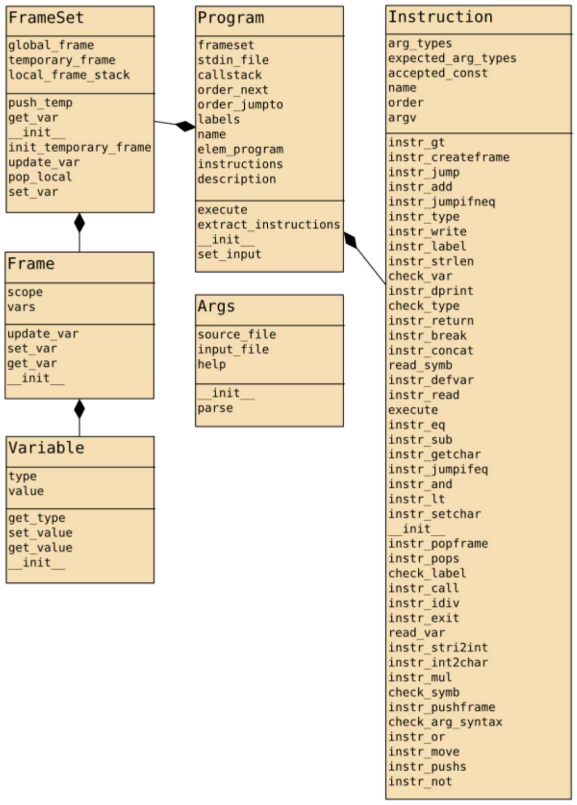
Documentation of Project Implementation for IPP 2018/2019 (interpret.py)

Name and surname: Michal Pospíšil
Login: xpospi95

Object Model

To better explain how this program works, let's establish that some of the classes create a layer structure. `Program` is the top layer and it is divided into smaller pieces with every layer. In the class diagram, class on the left from `Program` and lower in the layer model is `FrameSet` – wrapper for all frames that exist during the interpretation. Under `FrameSet` is `Frame` which consists of instances of class `Variable`. On the other side, class `Program` can be a top layer for class Instruction.



This layer separation works well for delegation of various checking that needs to happen before and during runtime. For example, updating variables is done with method of `FrameSet` which strips the frame specifier from the variable name, then calls method of that frame. `Frame` handles the updating operation with checking errors – which happens in one place and all instructions can rely that that variable exists in its context.

During implementation, a decision was made – every instruction is invoked by a method `Instruction.execute()`. It sounds logical, but in hindsight, better would be to have these methods in class `Program` since they manipulate with frames that can be referenced more easily from there. This was solved by passing a reference to `Program` instance into every `instr_*` function – these execute the actual task of every IPPcode19 instruction.

Implementation Details

As in the previous part, no design patterns were knowingly used. Technically, some methods can classify as decorators – `Instruction.check_arg_syntax()` and function `decode_escapes()`. Interpreter design pattern was considered but due to time constraints, own implementation was chosen.

Syntax checking

Firstly, source code represented in XML is checked. Use of `xml.etree` library made strictly adhering to the specification easier. To check that no other attributes or elements were used, all returned values were popped from list – if no exception is raised, XML is valid.

There was a discussion on supporting out-of-order and non-consecutive order attributes on the project forum. I decided to support both options even with negative values. Instructions are saved in a dictionary, where keys are their order attributes and values are instances of Instruction. In this way, control flow loop implemented in `Program.execute()` can easily sort the keys and find the next key. Next key is saved in Program so jump instructions can affect the control flow without returning anything.

There was an attempt to use an XSD schema to validate the XML with `lxml` library, but the fact that it is not installed on Merlin invalidated this option. Note on Wiki says that the library must be supplied with the script, but that was also not possible since it is not just a Python module and depends on some pure C code. Compilation of the module on Merlin was unsuccessful as other libraries were missing.

Arguments' syntax is checked when instructions are instantiated. Existence of labels in instructions is not done before runtime because no checking is done when the whole XML is parsed – that makes checking labels in later instructions impossible.

Escape sequences

It was surprising to find out that escape sequences in strings get escaped again by `xml.etree`. This was solved by replacing double backslashes followed by three digits with the actual character and saving that version of a string.

Error handling

Errors are printed on stderr and contain order attribute of instruction that is executed to guide the user. Script execution is immediately ended with `sys.exit()`. Better approach would be to use exceptions and this project showed that this it would benefit readability – since some functions couldn't return a value, they had to get the order attribute as an argument and that made them less readable.