Implementační dokumentace k 2. úloze do IPP 2018/2019

Jméno a příjmení: Radovan Babic

Login: xbabic09

Motivation

The second part of a project for subject IPP includes an interpret for a non-structured imperative language IPPcode19 implemented in `Python3` and a testing script implemented in `PHP7.3`. Both are also using a parsing script `parse.php` from previous part of this project.

My version of interpret is just MINIMAL implementation and it doesn't contain the test.php script!

Interpret `interpret.py`

The interpret for the IPPcode19 requires a source code parsed by `parse.php` as an XML file. To parse this file into an array of instructions, the interpret is using a XML parsing library `xml.etree.ElementTree`. The input XML file is defined by **required** argument `--source=file`. This file is then loaded, parsed and each instruction's syntax is checked for possible inconsistencies. If the input XML file is syntactically wrong, the interpret is exited with an adequate error code. If the syntax is correct, the interpreting part takes place.

Interpreting

The interpreting part consists of a main class `Instructions`, a final state machine like function `runProgram()` and a few helper functions for further checking, variable defining and more.

`Global frame(GF)` and `Labels` are implemented as a dictionary, which, in case of **GF,** holds the information about a variable name and its value ("Undefined" in case of just initialized variable without any concrete value) and in case of **Labels,** it holds the information about the label name and an instruction order number, where this label is located.

Running the interpret

Python3 interpret.py −−source=file.xml

Above you can see a prototype of the interpret running in a Linux terminal. When the interpret is executed, the input XML file is parsed (as mentioned above) and checked for inconsistencies. In case of a successful check, the instructions are parsed into an array and this array is sorted by the order number, which is included with every instruction (from 1 to N). After that, the `runProgram()` function is executed.

This function is iterating over the array using an **instruction pointer** that indicates which instruction is currently being interpreted. When an **operating code** (opcode) of the current instruction matches one of the implemented instructions, an adequate method from the `Instructions` class is invoked.

When the method is invoked, it starts interpreting the instruction, what also includes a semantic check. In case of a semantic error, the interpret is exited with an adequate error code. If all instructions have been interpreted successfully, the interpret is exited with code 0.

Implemented instructions

`DEFVAR`

Instruction for a variable initialization. The variable is initialized into the **GF** to `"Undefined"` value. To define a variable, a **MOVE** instruction has to be invoked.

`MOVE`

This instruction takes two parameters **WHERE** and **WHAT**. The WHERE parameter must be the variable, but WHAT can also be a symbol (int, string, bool, nil). Then the symbol WHAT is moved into the variable WHERE.

`WRITE`

This instruction takes one parameter which can be the variable or the symbol. This parameter is then printed onto a standard output.

`LABEL`

This instruction takes one parameter that consists only of **LETTERS** and is describing a line of code to which one of a **JUMP** instruction can be performed.

`JUMP`

The parameter of this instruction is the **LABEL**. When this instruction is interpreted, the instruction pointer is set according to LABELS's location.

`JUMPIFEQ`

The parameters of this instruction are the **LABEL** and the two **SYMBOLS** to be compared. If they are equal, the same logic is used as in the **JUMP** instruction.

`JUMPIFNEQ`

Same as above, but **JUMP** is performed only if the **SYMBOLS** are not equal.

`EXIT`

Exits the interpret with a code specified by the parameter with an **INTEGER** value.

`ADD/SUB/MUL/IDIV`

Arithmetic operations for the integer values. The first parameter is the **SYMBOL** to which the result of an operation is stored and the next two are the **INTEGER** values on which the selected operation is to be performed.