Implementation documentation for the 2nd task in IPP 2022/23 university course

Author: Alina Vinogradova
Login: xvinog00

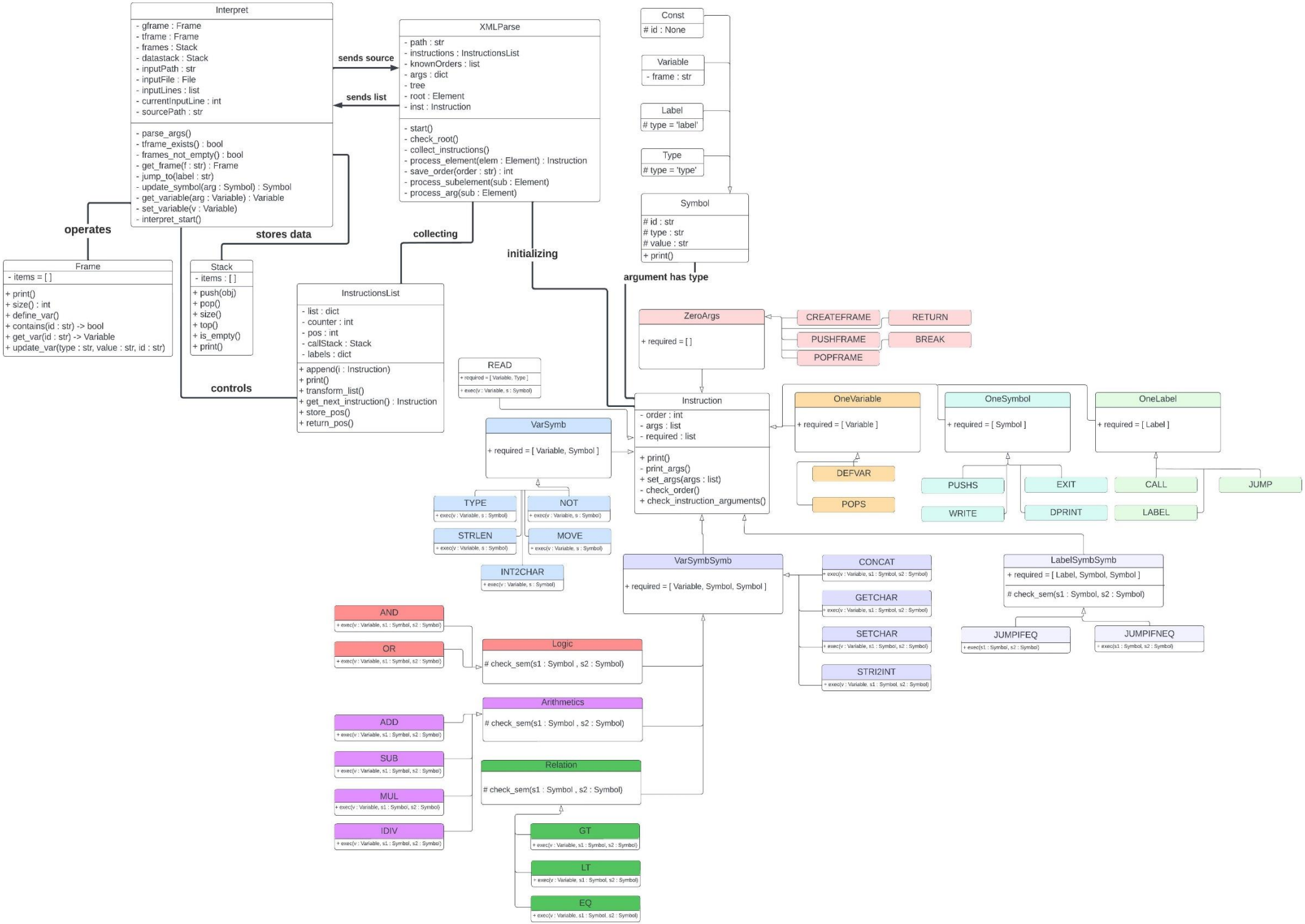
interpret.py script

The script was implemented using object-oriented programming in the `Python` language. The most basic and main class that represents the project is `Interpret`, which connects all the other implemented modules.

After running the program, the first method called is the `parse_args()` method, which processes the arguments given with the script. If successful, the data from these arguments will be collected: the file with the source XML code and/or the file with the input for the interpreter.

Next, the main program method `interpret_start()` is called, which first creates an instance of `XMLParse` class, passing during initialization data about where the file with XML code is located. The parser checks if the structure of the XML tree is correct, then processes the tree one instruction at a time, checking it and its operands (in the context of the tree element). In case no errors are found, the parser creates an `Instruction` class instance, more specifically its subclass, according to this instruction and then writes the created object to the `instructions` attribute, which is an instance of the `InstructionsList` class, and which is subsequently used by the main interpret program.

After all the raw data have been successfully processed and transformed into the necessary objects for the interpretation, the script processes the objects from the `InstructionsList` by calling the `get_next_instruction()` method within the `while` loop until it reaches the end of the list (in my implementation, in this case method returns `None`), successively executing each instruction and updating frames and possible variables within them, as well as call stack, data stack and other attributes responsible for the current state and the future progress of the interpretation.


Interpret
- gframe : Frame
- tframe : Frame
- frames : Stack
- datastack : Stack
- inputPath : str
- inputFile : File
- inputLines : list
- currentInputLine : int
- sourcePath : str
- parse_args()
- tframe_exists() : bool
- frames_not_empty() : bool
- get_frame(f : str) : Frame
- jump_to(label : str)
- update_symbol(arg : Symbol) : Symbol
- get_variable(arg : Variable) : Variable
- set_variable(v : Variable)
- interpret_start()
sends source
sends list
XMLParse
- path : str
- instructions : InstructionsList
- knownOrders : list
- args : dict
- tree
- root : Element
- inst : Instruction
- start()
- check_root()
- collect_instructions()
- process_element(elem : Element) : Instruction
- save_order(order : str) : int
- process_subelement(sub : Element)
- process_arg(sub : Element)
Const
id : None
Variable
- frame : str
Label
type = 'label'
Type
type = 'type'
Symbol
id : str
type : str
value : str
+ print()
operates
stores data
collecting
initializing
argument has type
Frame
- items = [ ]
+ print()
+ size() : int
+ define_var()
+ contains(id : str) -> bool
+ get_var(id : str) -> Variable
+ update_var(type : str, value : str, id : str)
Stack
- items : [ ]
+ push(obj)
+ pop()
+ size()
+ top()
+ is_empty()
+ print()
controls
InstructionsList
- list : dict
- counter : int
- pos : int
- callStack : Stack
- labels : dict
+ append(i : Instruction)
+ print()
+ transform_list()
+ get_next_instruction() : Instruction
+ store_pos()
+ return_pos()
ZeroArgs
+ required = [ ]
CREATEFRAME
RETURN
PUSHFRAME
BREAK
POPFRAME
READ
+ required = [Variable, Type]
+ exec(v : Variable, s : Symbol)
Instruction
- order : int
- args : list
- required : list
+ print()
- print_args()
+ set_args(args : list)
- check_order()
+ check_instruction_arguments()
OneVariable
+ required = [Variable]
OneSymbol
+ required = [Symbol]
OneLabel
+ required = [Label]
VarSymb
+ required = [Variable, Symbol]
DEFVAR
POPS
PUSHS
WRITE
EXIT
DPRINT
CALL
LABEL
JUMP
TYPE
NOT
STRLEN
MOVE
INT2CHAR
+ exec(v : Variable, s : Symbol)
VarSymbSymb
+ required = [Variable, Symbol, Symbol]
CONCAT
GETCHAR
SETCHAR
STRI2INT
+ exec(v : Variable, s1 : Symbol, s2 : Symbol)
LabelSymbSymb
+ required = [Label, Symbol, Symbol]
# check_sem(s1 : Symbol, s2 : Symbol)
JUMPIFEQ
JUMPIFNEQ
+ exec(s1 : Symbol, s2 : Symbol)
AND
OR
Logic
check_sem(s1 : Symbol , s2 : Symbol)
ADD
SUB
MUL
IDIV
Arithmetics
check_sem(s1 : Symbol , s2 : Symbol)
Relation
check_sem(s1 : Symbol , s2 : Symbol)
GT
LT
EQ