Documentation of Project 2 Implementation for IPP 2020/2021
Name and surname: Ondřej Ondryáš
Login: xondry02

# Interpreter

The interpreter loads an IPPcode21 program in its XML representation, checks the validity of the input XML and processes the instructions one-by-one. Instructions are represented by instances of subclasses of the `Instruction` class. These *stateless* subclasses implement the logic of the individual instructions. This logic operates on an object of the `InterpreterContext` type, which represents the current state of execution. Unexpected states are handled using custom exceptions. The `FLOAT` and `STACK` extensions have been implemented as well.

## Interpreter structure

The interpreter program is divided into several modules. The entry point is located in the `interpret` module. The `main` function parses the command-line arguments and creates an instance of the `Interpreter` class. It loads the whole input XML file into memory (using the `xml.etree` module), iterates through it and checks whether all the elements are valid according to the format specification[1]. This includes a check of whether the number of arguments is correct for the instruction and, in the case of constants, parsing the literal value according to the type of the argument, to detect malformed literals. Then, a corresponding instruction definition object (see Instruction definitions) is retrieved and its processing method is executed.

The `data_structures` module contains definitions of several data structures used during interpretation. When interpretation begins, an instance of `InterpreterContext` is created. This object represents the state of execution and may be seen as a 'virtual memory' of the interpreted program, as it contains stacks, frames with variables, label targets, program counter and other flags. It also provides an interface for stack operations (push/pop) over the variable stack and the frame stack, as well as helper methods for executing jumps.

Variables, as well as constants (literal values), are represented by instances of the `Variable` class. `Variable` objects are created when processing instruction arguments. More specifically, program variables are encapsulated by `Frame` objects that are stored in and managed by the context. This way, the variable lookup process can simply retrieve the correct frame object (the context provides the current top of the frame stack) and call its `get_var` or `def_var` method. For constants, a temporary `Variable` object is created and passed to the instruction process method. `Variable` objects are also used as containers for labels. This way, such objects can represent every possible value used as an instruction argument. Literals (numeric or string) from the source XML are parsed using static methods of the `LiteralParser` class when processing the argument that contains them.

All interpretation error states are handled using exceptions. These may be raised at any point during execution and they are caught at the top level, in the main method. They are defined in the `exceptions` module. All of them define an exit code that is returned by the program.

## Instruction definitions

The actual interpretation logic of each instruction is represented by subclasses of the `Instruction` abstract class, located in the `instruction_definitions` module. In its constructor, the base class receives a list of `ArgumentDefinition` objects that hold information about possible types of variables that the instruction accepts as its arguments. A `check_arguments` method is implemented in the base class that takes a list of variables and checks whether it matches the instruction's argument types. Subclasses then implement the `process` abstract method which takes a context and a list of variables representing the arguments and performs the instruction's logic over them.

Instances of the instruction subclasses are stored in a global list `instruction_defs` which is accessed when processing the program's instructions. This means that the instances of `Instruction` subclasses must behave statelessly regarding the executed program – the execution state gets passed to them only when their `process` method is called. Most instructions do not require complex logic and may be represented using a simple expression. For that reason, an `InlineInstruction` implementation is provided. It takes a callback to a process method in its constructor which may be passed a lambda expression.

Stack instructions are implemented using a simple wrapper `Instruction` subclass that encapsulates another `Instruction` object and assembles the argument list for its execution by popping the correct number of variables from the context's variable stack.

[1] This is done completely 'by hand' and could be improved by using mechanisms such as XSD validation.

### Jumps

This interpreter was designed to be one-pass[2]. The context contains a label cache that is empty when interpretation starts. When a `LABEL` instruction is processed, an entry is created in the cache. However, when performing a ‘forward’ jump instruction, the corresponding label may not have been resolved yet. For this reason, a ‘current jump target’ flag has been added to the context. When it is set, the main interpreter loop goes on but does not invoke any instruction processing methods except for `LABEL` instructions, until the corresponding label or end of the input program is reached. This process is also always speculatively invoked for conditional jumps `JUMPIF[N]EQ` (even if the jump is not actually performed), as a part of semantic check of the label’s existence. In this case, an additional ‘performing lookup jump’ flag is set in the context to signalise that after finding the label, program counter should be set back to its original value.

## Testing script

The `parse.php` script also uses an object model to achieve certain level of separation of concerns. It also includes the `FILES` extension, as well as some custom extensions that improve its usability.

The primary, test-running functionality is implemented in the `TestSuite` class. It encapsulates a single *test*[3] together with its execution state and result. Its public interface contains getter methods for different aspects of its execution result and, most importantly, the `run` method which executes the individual programs and compares their outputs to the expected ones.

A collection of finished test suites is later processed by a `ReportAssembler` instance. This class implements a very basic templating engine that loads an HTML template, extracts predefined sections and replaces placeholders with results of the test executions.

At the beginning, the script parses command line arguments and creates a `Config` object which carries the options to the other components. Error states are handled using a custom `TestException`, which can be generated when running a test suite and is propagated to the main loop, where it’s handled.

Extra command line parameters can be used in addition to the ones required by the assignment:

- `--capture-output` enables saving output of the executed programs (parser, interpreter and the diff tools) and including it in the generated report.
- `--status` prints information about test running progress to `stderr`.
- `--outfile=[file]` saves the report to a file instead of putting it to `stdout`.

[2]Adding input program format checking introduced a two-pass aspect to it. However, it is desirable not to ‘bloat’ this phase by performing additional complex semantic checks and actions.

[3]In this context, *test* means a single input `.src` file, with optional additional files that contain expected output, input or exit code.