Assignment of a project from the subject IPP 2023/2024

Zbynўk Kўivka, Radim Kocman E-
mail: krivka@fit.vut.cz

1 Basic characteristics of the project

Design, implement and document two programs for parsing and interpreting the unstructured imperative language IPPcode24. Create the corresponding brief program documentation for the implementation. The project consists of two tasks and is individual.

The first task consists of a Python 3.10 program (see Section 3) run using the parse.py script and the documentation for this program (see Section 2.1). The second task consists of a PHP 8.3 program (see section 4) integrated into the provided framework (run by interpreter.php) and documentation for your implementation (see section 2.1 and details in section 4).

2 Requirements and organizational information

In addition to the implementation of both programs and the creation of documentation, a number of the following formal requirements must also be observed. If one is not followed, the project can be evaluated with zero points! To check at least some formal requirements, the is_it_ok.sh script available in *the Mo-odle* of the IPP subject can be used.

Terms:

comments[1] on the assignment of the project by February 13, 2024;

fixation of the project assignment from February 14, 2024;

submission of the first task on Tuesday, **March 12 ,** 2024 by 23:59:59;

submission of the second task on Tuesday, **April 16 ,** 2024 by 23:59:59.

Additional information and consultation on the project from IPP:

- *E-Learning* (Moodle) of the IPP subject including Frequently Asked Questions *(FAQ)* and Files to pro-
  throw up
- *Forum* in *E-Learning* (Moodle) IPP for ac. year 2023/2024, topics Project.*.
- Zbynўk Kўivka (project guarantor): according to consultation slots (see website) or by agreement by e-mail (indicate the subject beginning with "IPP:"), see http://www.fit.vut.cz/person/krivka.
  You can find other trainers on the subject card.
- Dušan Koláў (subject guarantor; only in serious cases): by agreement by e-mail (indicate the subject beginning with "IPP:"), see http://www.fit.vut.cz/person/kolar.

[1] If you find an error or ambiguity in the assignment, please let us know on the Subject *Forum* or by e-mail at krivka@fit.vut.cz. Valid comments and error alerts will also be awarded with bonus points.

If you have any questions, problems or ambiguities regarding this project, after reading *the FAQ* and *the Forum* (also use the search options on *the Forum),* do not hesitate to write to *the Forum* (for a general problem that potentially affects your colleagues as well) or contact the project guarantor, trainer (in the case of an individual problem), or the subject guarantor in an emergency, when always put the string "IPP:" at the beginning of the subject. Problems detected in the order of hours to a few days before the deadline for submitting a part of the project will not be taken into account. Therefore, start solving the project well in advance.

Form and method of submission: Each assignment is submitted individually through StudIS to the IPP subject (submission by e-mail is subject to points), and by submitting you confirm the exclusive authorship of the scripts and documentation.

In the activity "Project - 1st task in Python 3.10" you submit the archive for the first task (parse.py script; including the documentation of this script). After the deadline for submitting the 1st task, the submission of the archive will be opened in the activity "Project - 2nd task in PHP 8.3" for the second task (contents of the student directory within *ipp-core;* including documentation).

Each task will be submitted in a special archive, where the files for the given task will be compressed with the ZIP, TAR+GZIP or TAR+BZIP program into a single archive named xlogin99.zip, xlogin99.tgz, or xlogin99.tbz, where xlogin99 is your login. The size of each archive will be limited by the information system (probably to 2 MB). The archive must not contain special or binary executable files, nor the *ipp-core framework provided by us.* The names of all files and directories can only contain letters of the English alphabet, numbers, a period, a dash and an underscore (do not include hidden directories, which typically begin with a period). **The job scripts (or, in the case of the 2nd job, the contents of the student folder) will be located in the root directory of the submitted archive.** After unpacking the archive on the *Merlin* server (in the case of the 2nd task, we unpack the archive into the *student* directory of our *ipp-core* framework) , it will be possible to run the script(s). The archive may contain a reasonable number of directories (typically for your modules/namespaces, custom tests and helper scripts, or enabled libraries2 that are not installed on the *Merlin server).*

Evaluation: The amount of the basic point evaluation of the project in the IPP subject is **a maximum of 20 points.** In addition, a maximum of 5 bonus points can be obtained for a high-quality and successful solution to one of the extensions or qualitatively above-average participation in the project *Forum ,* etc.

Evaluation of individual scripts: parse.py up to 7 points (of which up to 1 point for documentation and modification of the source codes of the parse.py script); interpreter.php up to 13 points (of which the design, documentation and editing of source texts of scripts up to **4 points).** However, the documentation is evaluated at a maximum of 50% of the sum of the evaluation of the functionality of the task (i.e. if no functional script is submitted in the given task, the documentation itself will be evaluated with 0 points). The points for each task, including its documentation, are rounded to tenths of a point before being entered into StudIS.

The scripts will be run on the *Merlin* server with the command: *interpreter script parameters,* where *the interpreter* will be python3.10 or php8.3, *the script* and *parameters* depend on the task. The evaluation of most functionality will be provided by an automated tool. The quality of the documentation, comments, design and structure of the source code will be evaluated by the trainees.

Your scripts will be evaluated independently of each other, so it is possible to submit, for example, only the 2nd assignment without submitting the 1st task.

The conditions for repeating students regarding the possible recognition of last year's project assessment can be found in *the FAQ* on the course's *Moodle .*

2 For the 2nd task, the enabled libraries will be available in the vendor folder.

Registered extensions: In the case of implementing some registered extensions for bonus points, the submitted archive will contain an extension file in which you indicate the identifier of one implemented extension3 on each line (the lines are terminated by a Unix line end, i.e. a character with a decimal ASCII value of 10). During the solution, new job extensions can be registered for bonus points (see *Forum).* You can send proposals to *the Forum* for new non-trivial extensions that you would also like to implement, no later than the deadline for the trial submission of the given task . The trainee decides to accept/not accept the extension and evaluate the extension according to its difficulty, including the assignment of a unique identifier. Implemented extensions not identified in the extensions file will not be evaluated.

Trial submission: To increase students' motivation for completing tasks on time, we offer the concept of optional trial submission. In exchange for a trial submission by the specified deadline (approx. a week before the final deadline), you will receive feedback in the form of inclusion in one of five evaluation ranges (0–10%, 11–30%, 31–50%, 51–80%, 81 –100%). If your trial submission is within the first rating range, you have the option to personally consult the reason if you don't discover it yourself.
For other ranges, more detailed information will not be provided.

The trial submission will be evaluated relatively quickly by automatic tests and indicative information will be sent to the students about the correctness of the trial submission task from the point of view of part of the automatic tests (i.e. it will not be a final assessment; therefore neither points nor a more precise percentage assessment will be communicated). The use of the trial period is not mandatory, but its non-use may be negatively taken into account in the event of a project evaluation complaint.

The formal requirements for trial submission are identical to the requirements for the final deadline, and the submission will be made to the special activities "Project - Trial submission of the 1st task" until **March 5, 2024** "Project - Trial submission of the [a] 2nd task" until **April 9, 2024.** Not must include documentation,
which together with the extensions will not be experimentally evaluated.

2.1 Documentation

The implementation documentation (hereinafter referred to as the documentation) must be a brief and comprehensive guide **to your way of solving** scripts 1., respectively. 2. tasks. It will be created in **PDF** or **Markdown** format (see [4]). Any documentation formats other than PDF or Markdown4 (md extension) will be ignored, which will result in the loss of documentation points. The documentation can be written either in Czech, Slovak (with diacritics, formally purely) or English (formally purely).

The documentation will describe the overall design philosophy, internal representation, method, and your specific solution procedure (e.g. resolution of questionable cases insufficiently specified by the specification, specific extension solution, possible use of design patterns, implemented/unfinished features). It is advisable to supplement the documentation with, for example, a UML class diagram (obligatory for the 2nd task), a designed finite state machine, the rules of the grammar created by you, or a description of other formalisms and algorithms. However, **it may not contain even a partial copy of the assignment.**

The planted documentation of the 1st task describing the parse.py script should not exceed 1 A4 page (ie the range of approximately 1 to 2 standard pages). For the 2nd task (describing the interpreter.php script), do not exceed the planted 2 pages of A4 (not including a suitably large UML diagram). Typesetting recommendations: 10-point Times New Roman for text and Courier for identifiers and really short snippets of interesting code (or back apostrophes in Markdown); do not include any special cover page, table of contents, or conclusion.
It is reasonable to use first and possibly second level headings (12-point and 11-point Times New Roman or ## and ### in Markdown) to create a logical structure of the documentation.

[3] Extension identifiers are listed in bold for the specific extension.

4 If both Markdown and PDF documentation is present, the PDF version will be evaluated, where the rate is more certain.

The title and header of the documentation5 will contain on the first three lines:

Implementation documentation for %cislo%. assignment to IPP 2023/2024 Name
and surname: %name_surname% Login:
%xlogin99%

where %name_surname% is your first and last name, %xlogin99% your login and %cislo% is the number of the documented task.

The documentation will be in the root directory of the submitted archive and named readme1.pdf or readme1.md for the first task and readme2.pdf or readme2.md for the second task.

Functional/object decomposition and commenting of source codes will also be evaluated as part of the documentation. If it is not crystal clear from the function/module/class/parameter identifier itself and any type annotation/help, add a suitable comment about the purpose, etc.

2.2 Program Part

Project entry requires the implementation of two scripts6 that have command line parameters and define how inputs and outputs are handled. The script must not run any other processes or operating system commands. Direct all error messages, warnings and debugging statements only to the standard error output, otherwise you will probably not comply with the specification due to the modification of the defined outputs (either to external files or to the standard output). If the script runs without errors, a return value of 0 (zero) is returned. If an error occurs, an error return value greater than zero is returned. Errors have binding error return values:

- 10 - missing script parameter (if necessary) or use of a prohibited combination of parameters;
- 11 - error when opening input files (eg non-existence, insufficient permissions);
- 12 - error when opening output files for writing (eg insufficient permissions, error during enrollment);
- 20 – 69 - error return codes specific to individual scripts;
- 99 - internal error (not affected by integration, input files or command line parameters).

Unless specified otherwise, all inputs and outputs are in UTF-8 encoding. For the purposes of the IPP project, the default locale7 setting must be left on the *Merlin* server , i.e. LC_ALL=cs_CZ.UTF-8.

The names of the main scripts are given by the input. Helper scripts or libraries will have the extension according to the custom in the given programming language (.py for Python 3 and .php for PHP 8). Script evaluation will be performed on a *Merlin* server with current versions of interpreters (python3.108 version 3.10.11 and php8.39 version 8.3.2 were installed on this server on 2/1/2024).

5 The English version of the title and header of the documentation can be found in *the FAQ* on *Moodle* subject
6 These scripts are command-line applications, or console applications.
7Correct environment settings are necessary to use and correctly process command line parameters in UTF-8. For proper functionality, the encoding of the console character set must also be set to UTF-8 (for example, with the PuTTY program in the *Window.Translation* category, you set *the Remote character set* to UTF-8). For a change affecting the current session, the unix command export LC_ALL=cs_CZ.UTF-8 can be used.
8Warning: Python3.10 command testing must be followed on *Merlin* server as python only runs old incompatible version! Python 3.x is not backwards compatible with version 2.x!
9Warning: On the *Merlin* server , testing with the php8.3 command must be followed, because only php runs the version, which has limited access to the file system!

The standard pre-installed libraries of both language environments on the *Merlin server can be used for the solution.* The possible use of another library must be consulted with the project guarantor (primarily for the reason that the solution of the project using a suitable library does not become completely trivial and, if necessary, the library is available within *ipp-core).* The list of permitted and prohibited libraries is kept up-to-date on *Moodle.* In PHP scripts, some functions are disabled for security reasons (e.g. header, mail, popen, curl_exec, socket_*; a complete list is in the enabled/disabled libraries on *Moodle).*

Each script will work with one common parameter:

- --help writes help to the standard output of the script (does not load any input), which can be taken from the input (diacritics can be removed, or translated into English according to the selected language of the documentation), and returns the return value 0. This parameter cannot be combined with any other parameter , otherwise exit the script with error 10.

Combinable script parameters are separated by at least one whitespace character and may be listed in any order unless otherwise specified. It is also possible to implement your own non-collision parameters for scripts (we recommend consulting on *the Forum* or with the project guarantor).

If not stated otherwise, according to the conventions of Unix systems, short (one-hyphen) and long (two-hyphen) substitute parameters can be considered, which can be interchanged while preserving the semantics (so-called alias parameters), but the long versions will always be tested.

If a file (eg --source=file or --source="file") or a path is part of the parameter, this file/path can be specified as a relative path10 or an absolute path; do not consider the occurrence of quotation marks and equal signs in *the file .* Paths/filenames can also contain Unicode characters in UTF-8.

Recommendation: Since a large part of the evaluation is derived by automated tests, we recommend using your own automated testing for script development as well. For this, a set of tests is needed, which you compile on the basis of a correct understanding of the assignment. Several published tests can be inspired. In addition to the return code, it is advisable to test the correctness of the output in case of successful completion of the script.
Tests can also be shared between students.

3 Code parser in IPPcode24 (parse.py)

A filter type script (parse.py in Python 3.10) reads the source code in IPP-code24 from the standard input (see section 5), checks the lexical and syntactic correctness of the code and writes to the standard output an XML representation of the program according to the specification in section 3.1.

This script will work with the following parameters:

- --help see common parameter of all scripts in section 2.2.

Parser-specific error return codes:

- 21 - wrong or missing header in the source code written in IPPcode24; • 22 - unknown or incorrect operation code in the source code written in IPPcode24; • 23 - other lexical or syntactic error of the source code written in IPPcode24.

10 The relative path will not contain the ~ (tilde) wildcard.

3.1 Description of the output XML format

The mandatory XML header11 is followed by the root element program (with the mandatory text attribute language with the value IPPcode24), which contains instruction elements for instructions. Each instruction element contains a mandatory order attribute with the order of the instruction. When generating elements, the order is numbered from 1 in a continuous sequence. Furthermore, the element contains the mandatory opcode attribute (the opcode value is always in capital letters in the output XML) and elements for the corresponding number of operands/ arguments: arg1 for the possible first argument of the instruction, arg2 for the possible second argument and arg3 for the possible third argument of the instruction. The argument element has a required type attribute with possible values of int, bool, string, nil, label, type, var depending on whether it is a literal, label, type, or variable, and contains a text element.

This text element then carries either the value of a literal (no longer specifying the type and without the @ sign), the label name, the type, or the variable identifier (including specifying the memory frame and @). For variables, always write the name of the memory frame in capital letters, as it should already be on the input.
Leave the case of the variable name itself unchanged. The format of integers is decimal, octal or hexadecimal according to Python 3 conventions, however, output these numbers exactly in the format in which they were read from the source code (e.g. positive number signs or leading excess zeros will remain). For string literals when writing to XML, do not convert the original escape sequences, but only use the corresponding XML entities (e.g. <, >, &) for problematic characters in XML (e.g. <, >, &). Similarly, convert problematic characters occurring in variable identifiers. Always write bool literals in lowercase like false or

true.

Recommendation: Note that the analysis of IPPcode24 is so-called context-dependent (see lectures), where for example you can have a keyword used as a label and from the context it is necessary to recognize whether it is a label or not. When creating the analyzer, we recommend combining finite-state control and regular expressions and using a suitable library to generate the output XML.

The output XML will be compared with the benchmark results using the A7Soft JExamXML12 tool see [2]. Attention, in Python the return command is not used to return an error code, use the sys.exit function.

If you are just learning to write readable code in Python 3, we recommend studying [6].

3.2 Bonus Extensions

STATP Collection of processed source code statistics in IPPcode24. The script will support the parameter --stats=file to specify *a file* where the statistics will be written in the parameters placed after this --stats. Statistics are written to the file line by line according to the order in the parameters with the possibility of repeating them; write nothing on each line except the desired numeric output and line breaks; possibly an existing file is overwritten. To collect groups of statistics into different files, another occurrence of the --stats parameter with a different file name is used, followed by other parameters of the new statistics group. The --loc parameter lists the number of lines with instructions in the statistics (empty lines or lines containing only a comment or an introductory line are not counted). The --comments parameter lists the number of lines on which the comment occurred in the statistics. The --labels parameter lists the number of defined labels (i.e. unique possible jump targets) in the statistics. The --jumps parameter lists the number of all call return instructions and *jump* instructions (sum of conditional/unconditional jumps and calls), --fwjumps the number of forward jumps, --backjumps the number of backward jumps, and --badjumps the number of jumps

11Traditional XML header including version and encoding is <?xml version="1.0" encoding="UTF-8"?> 12A7Soft JExamXML settings for XML comparison (options file) are on Moodle.

on a non-existent sign. If only the --stats parameter is specified without specifying the statistics to be listed, the output will be an empty file. The --frequent parameter lists in the statistics the names of operation codes (in capital letters, separated by a comma, without spaces, alphabetically ascending), which are the most frequent in the source code according to the number of static occurrences. The parameter --print=string prints the *string string* to the statistics , and the parameter --eol prints the line breaks to the statistics. If --loc, --comments and other statistics parameters are specified without the --stats parameter before them, this is error 10. Attempting to write multiple groups of statistics to the same file during one script run results in error 12. [1.5b]

NVP When designing and implementing the parse.py script and auxiliary scripts, object-oriented programming will be applied and at least one suitable standard design pattern will be used, with the exception of Jedináÿka (see [5] and tips *on Moodle).* A condition for evaluating this extension is proper documentation (why, where, how, description of limitations). If the extension is mentioned in the documentation and without a serious effort to implement it, it can receive a one-point malus (i.e. -1 b). [1b]

4 Interpreter XML code representation (interpret.php)

Familiarize yourself with the supplied *ipp-core* framework in PHP 8.3 and, with its **mandatory** use, design and implement an interpreter that loads the XML representation of the program and this program interprets and generates output using the input according to the command line parameters. The supplied ipp-core framework will help with some basic operations *(e.g.* loading standard input, listings, basic processing of parameters).
The input XML representation is defined a little more loosely than the XML generated by the parse.py script, but it can be assumed that the input XML representation will no longer contain the errors13 that parse.py was tasked with detecting. Since the *ipp-core* framework is designed in an object-oriented way, we require that its use and expansion be done in an object-oriented way, considering the use of suitable design patterns (recommended on Moodle). For object design and implementation, it is of course mandatory to also write a brief and terminologically correct documentation of how you used/extended the *ipp-core* framework to fulfill the assignment.

In addition, compared to section 3.1, the interpreter supports the existence of optional documentation text attributes name and description in the root program element and differently formatted tags (e.g. optional use of abbreviated tag notation if the tag does not contain a subelement). The semantics of single IPPcode24 instructions is described in section 5. The interpretation of instructions takes place according to the order attribute in ascending order (however, the sequence does not have to be continuous/ordered unlike section 3.1).

This script will work with the following parameters:

- --help see common parameter of all scripts in section 2.2;
- --source=file input file with an XML representation of the source code according to the definition from section 3.1 and additions at the beginning of section 4;
- --input=file file with inputs14 for the actual interpretation of the entered source code.

At least one of the parameters (--source or --input) must always be specified. If one of them is missing, the missing data is read from standard input.

13However, the generated output XML is not otherwise validated in the 1st task (see error return codes).
14Input/input file can be empty; e.g. if no READ instruction is interpreted.

Interpreter-specific error return codes:

- 31 - incorrect XML format in the input file (the file is not well *-formed,* see [1]); • 32 - unexpected XML structure (eg element

for argument outside element for instruction, instruction with duplicate order or negative order); • 88 - integration error (invalid integration with *ipp-core framework).*

Interpreter error return codes in the event of an error during interpretation are given in the IPPcode24 language description (see section 5.1).

We will also check your source codes of the second task by static analysis using the PHPStan tool, where in order to gain some points from the entire task, we require the fulfillment of level 0, and in order to gain a whole 1 point (part of the evaluation of the documentation and quality of the task code) without losses, we require the fulfillment of level 615, but we generally recommend level 9, which is the default in the tool.

The ipp-core **framework :** To support and illustrate the suitability of the object design, a single-minded framework named ***ipp-core is prepared,*** which you must use when solving this task. The framework is written in PHP 8.3 and its current version is available on the faculty git server16 or in the directory /pub/courses/ipp/ipp-core on the Merlin server. In the assignment, we only mention the basic features of the framework, you can find more information in the README and by studying the commented source code. It is advisable to discuss the ambiguities about the framework in the Forum. The *ipp-core* framework adheres to the PSR-417 class registration standard (autoloader). Installing a framework on a *Merlin* server to its own current directory might look like this:

```
git clone https://git.fit.vutbr.cz/IPP/ipp-core.git cd ipp-core php8.3
composer.phar
install
```

which installs scripts and configurations for *the autoloader* according to PSR-4 and the *PHPStan* tool to the vendor folder used for static code analysis and basic evaluation of the quality of your code. You then run the Level 6 analysis of your code on the *Merlin* server with the command:

```
php8.3 vendor/bin/phpstan analyze --level=6
```

Make all changes and create new files only in the student folder, the content of which you will be the only one to submit. I.e. documentation and any extensions must also be in this folder! The core folder holds the classes of the framework itself from the IPP\Core namespace, and the vendor folder may contain additional third-party libraries if they are enabled for use in this role.

The entire program using the framework is launched by the pre-prepared script interpreter.php. Your implementation starts in the student\Interpreter.php file and the corresponding Interpreter class in the IPP\Student namespace.

Use the *ipp-core* framework to load input code in XML format (DOMDocument is returned), basic processing of parameters (Settings class), and use the InputReader and OutputWriter interfaces to interpret input-output instructions. There will also be an is_it_ok.sh script for checking basic formal requirements and a Makefile with several basic targets. To print warnings to standard error output, interpreter.php contains the command ini_set('display_errors', 'stderr');.,

15https://phpstan.org/user-guide/rule-levels 16

We recommend that you log in for the first time via a web browser at https://git.fit.vutbr.cz/IPP in order to to automatically activate login as username instead of email address for SSH access. 17https://www.php-fig.org/psr/psr-4/

Documentation and object-oriented (OO) design: The code will mandatorily use the *ipp-core* framework and will be designed object-oriented, which means that you need to familiarize yourself with the OOP terminology (at least in the scope of lectures), think about the possibilities of using OO and design patterns for this task, describe the design briefly and clearly (with the correct terminology) in the documentation in accordance with your implementation (proposed but not implemented parts must be marked). In an appropriate way, project the OO design into the code (e.g. optional typing of method and function signatures) including the appropriate choice of identifiers of classes, attributes and methods with regard to self-documentation. If a one-word naming is not enough to clearly capture the purpose, add a comment, and for the method, describe the meaning/purpose of the parameters, if the name and type annotation are not enough. The UML class diagram is a mandatory part of the documentation, with the fact that the framework classes can be presented in abbreviated form (i.e. without attributes and methods), but for the sake of clarity, do not omit important relationships between classes and interfaces from the framework. However, appropriately differentiate the parts of the UML diagram that you have not implemented (e.g. with a gray font). Any use of design patterns must be properly justified and clearly documented how and by which classes it is implemented. Above all, the Jedináÿek design pattern is almost always used inappropriately and, moreover, poorly implemented.

Recommendation: In the case of an incomplete implementation, focus on the functionality of the global memory frame, work with variables of type int, WRITE instructions and instructions for controlling the program flow.

For development on a local machine, it is possible to take advantage of the fact that the *ipp-core* framework is available as a development container that can be conveniently used, for example, in Visual Studio Code (for more, see the README within the *ipp-core framework).*

4.1 Bonus Extensions

FLOAT Support for the float type in IPPcode24 (decadal and scientific notation in the analyzer and in the interpreter, including loading from standard input according to the rules for decimal numbers18 in PHP 8). Support instructions for working with this type: INT2FLOAT, FLOAT2INT, arithmetic instructions (including DIV), etc. (see [3]). Support in parse.py is possible but will not be tested. [1b]

STACK Support for stack variants of instructions (suffix S; see [3]): CLEARS, ADDS/SUB-S/MULS/IDIVS, LTS/GTS/EQS, ANDS/ORS/NOTS, INT2CHARS/STRI2INTS, and JUMPI-FEQS/JUMPIFNEQS. The stack versions of instructions select operands with input values from the data stack as described by the three-address instruction from the end (ie, typically first *ÿsymb2ÿ* and then *ÿsymb1ÿ).* Support in parse.py is possible but will not be tested. [1b]

STATI Collecting code interpretation statistics. The script will support the parameter --stats=file to specify a *file* where the aggregated statistics will be written (by lines according to the order in other (possibly repeated) parameters; do not write anything on each line except the desired numeric or string output). Support for the --insts parameter for listing the number of so-called executed instructions (ie, excluding debug instructions and special LABEL instructions) during interpretation into statistics. The --hot parameter writes the value of the order attribute to the statistics for the executed instruction that was executed the most times and has the smallest value of the order attribute. Support for the --vars parameter to list the maximum number of initialized variables present at one time in all valid memory frames during the interpretation of the specified program into statistics.

The --stack parameter writes to the statistics the maximum occupancy (number of stored values) of the data stack during the entire interpretation of the program. The parameter --print=string prints the *string string* to the statistics , and the parameter --eol prints the line breaks to the statistics. If the initial --stats parameter is missing when specifying one of the action parameters, it is error 10. [1 b]

18https://www.php.net/manual/en/language.types.float.php

5 Description of the IPPcode24 language

The unstructured imperative language IPPcode24 was created by modifying the IFJcode23 language (the language for the intermediate code of the IFJ23 translator, see [3]), which includes three-address instructions (typically with three arguments) and possibly stack instructions (typically fewer parameters and working with values on a data stack). Each instruction consists of an opcode (a keyword with the name of the instruction) that is case insensitive (i.e. case insensitive). The rest of the instructions consist of operands that are case sensitive. Operands are separated by any non-zero number of spaces or tabs. Any number of spaces or tabs can also appear before the opcode and after the last operand. Line breaks are used to separate individual instructions, so that there is a maximum of one instruction on each line and it is not allowed to write one instruction on more than one line. Each operand is made up of a variable, constant, type or label. Single-line comments starting with a hash (#) are supported in IPPcode24. With the exception of empty19 or commented lines, the code in the IPPcode24 language at the beginning instead of the instruction contains only the language identifier (a dot followed by the name of the language, where case does not matter):

```
.IPPcode24
```

5.1 Interpreter return values

If the interpretation takes place without errors, the return value given by the EXIT instruction or implicitly 0 (zero) is returned. The following return values correspond to error cases:

- 52 - error during semantic checks of input code in IPPcode24 (e.g. use of undefined of a tag, redefinition of a variable);
- 53 - runtime interpretation error – wrong types of operands; • 54 - runtime interpretation error – access to non-existent variable (memory frame exists); • 55 - runtime interpretation error – the memory frame does not exist (e.g. reading from an empty stack frames);
- 56 - runtime interpretation error – missing value (in a variable, on the data stack or in the call stack);
- 57 - runtime interpretation error – wrong operand value (e.g. division by zero, wrong return value of the EXIT instruction);
- 58 - runtime interpretation error – incorrect work with the string.

5.2 Memory model

During interpretation, we most often store values in named variables, which are grouped into so-called *memory frames,* which are essentially dictionaries of variables with their values.
IPPcode24 offers three kinds of memory frames:

- global, we denote GF (Global Frame), which is automatically initialized at the beginning of the interpretation as empty; used for storing global variables;
- local, we denote LF (Local Frame), which is initially undefined and refers to the top/current memory frame on the frame stack; serves for storing local variables of functions (the frame stack can be advantageously used when calling functions nested or recursively);

19A line containing only white characters is considered empty.

- temporary, we denote TF (Temporary Frame), which serves to prepare a new or clean up an old memory frame (e.g. when calling or completing a function), which can be moved to the frame stack and become the current local memory frame. At the beginning of the interpretation, the temporary memory frame is undefined.

Overlaid (previously inserted) local memory frames in the frame stack cannot be accessed before taking out later added frames.

Another option for storing unnamed values is a data stack used by stack instructions.

5.3 Data types

IPPcode24 works with operand types dynamically, so the variable type (or memory location) is given by the contained value. Unless otherwise stated, implicit conversions are prohibited. The interpreter supports the special value/type nil and three basic data types (int, bool, and string), whose ranges and precisions are compatible with PHP 8.

The notation of each constant in IPPcode24 consists of two parts separated by a wrapper (@ sign; no whitespace characters), an indication of the type of the constant (int, bool, string, nil) and the constant itself (number, literal, nil). E.g. bool@true, nil@nil or int@-5.

The int type represents an integer (don't handle overflow/underflow). The bool type represents a truth value (false or true). In the case of a constant, the literal for the string type is written as a sequence of printable characters in UTF-8 encoding (except white characters, hash (#) and backslash (\)) and escape sequences, so it is not delimited by quotation marks. The escape sequence, which is necessary for characters with the decimal code 000-032, 035 and 092, is of the form \xyz, where xyz is a decimal number in the range 000-999 consisting of exactly three digits20; eg a constant

string@ÿetÿzec\032with\032slash\032\092\032and\010new\035line

represents a string

a string with a backslash \ and
a new#line

An attempt to work with a non-existent variable (reading or writing) leads to error 54. An attempt to read the value of an uninitialized variable leads to error 56. An attempt to interpret an instruction with operands of unsuitable types according to the description of the given instruction leads to error 53.

5.4 Instruction Set

When describing the instructions, we put the operation code in bold and write the operands using non-terminal symbols (possibly numbered) in angle brackets. The non-terminal *ÿvarÿ* denotes a variable, *ÿsymbÿ* a constant or variable, *ÿlabelÿ* denotes a label. The variable identifier consists of two parts separated by a wrapper (@ character; no whitespace characters), the memory frame designation LF, TF or GF and the variable name itself (a sequence of any alphanumeric and special characters without whitespace characters starting with a letter or special character, where special characters are : _, -, $, &, %, *, !, ?). E.g. GF@_x denotes the _x variable stored in the global memory frame.

The same rules apply to the label entry as to the variable name (i.e. the part of the identifier after the suffix).

Example of a simple program in IPPcode24:

20 We will not test the writing of characters with a Unicode code greater than 126 using these escape sequences.

```
.IPPcode24
DEFVAR GF@counter
MOVE GF@counter string@ #Initialize variable to empty string
#Simple iteration                    until the specified condition is met
LABEL while
JUMPIFEQ end GF@counter string@aaa
WRITE string@Variable\032GF@counter\032contains\032
WRITE GF@counter
WRITE string@\010
CONCAT GF@counter GF@counter string@a
JUMP while
LABEL end
```

The instruction set offers instructions for working with variables in memory frames, various jumps, operations with a data stack, arithmetic, logical and relational operations, as well as conversion, input/output and debugging instructions.

5.4.1 Working with memory frames, calling functions

MOVE *ÿvarÿ ÿsymbÿ* Assign a value to a variable Copies the value of *ÿsymbÿ* to *ÿvarÿ.* E.g. MOVE LF@par GF@var copies the value of the variable var in the global memory frame to the variable par in the local frame.
CREATEFRAME Create a new temporary memory frame Creates a new temporary memory frame and discards any contents of the original temporary frame.
PUSHFRAME Move temporary frame to frame stack Move TF to frame stack. The memory frame will be available through the LF and will overwrite the original frames on the frame stack. The TF will be undefined after the instruction is executed and must be created with CREATEFRAME before it can be used again. Attempting to access an undefined frame results in error 55.
POPFRAME Move current memory frame to temporary Move peak LF frame from framestack to TF. If no frame is available in the LF, error 55 will occur.
DEFVAR *ÿvarÿ* Define new variable in memory frame Defines a variable in the specified frame according to *ÿvarÿ.* This variable is still uninitialized and without a type determination, which will only be determined by assigning a value. Repeated definition of a variable already existing in the given frame results in error 52.
CALL *ÿlabelÿ* Jump to label with return support Saves the current incremented position from the internal instruction counter to the call stack and jumps to the specified label (if any, other instructions must prepare the memory frame).
instruction Return to the position stored by the CALL **RETURN** Removes the position from the call stack and jumps to this position by setting the internal instruction counter (cleaning of local frames must be provided by other instructions). Executing the instruction when the call stack is empty results in error 56.

5.4.2 Working with the data stack

The opcode of stack instructions ends with the letter S". Stack instructions optionally load the missing operands from the data stack and store the resulting value of the operation back to the data stack.

PUSHS *ÿsymbÿ* Put the value on the top of the data stack Stores the value of *ÿsymbÿ* on the data stack.

POPS *ÿvarÿ* Pop a value off the top of the data stack
If the stack is not empty, it takes the value from it and stores it in the *ÿvarÿ variable,* otherwise an error occurs 56.

5.4.3 Arithmetic, relational, Boolean and conversion instructions

This section describes the three-address instructions for the classic operations for calculating an expression. Overflow or underflow of the numerical result, do not solve it.

ADD *ÿvarÿ ÿsymb1ÿ ÿsymb2ÿ* Sum of two numeric values
Adds *ÿsymb1ÿ* and *ÿsymb2ÿ* (must be of type int) and stores the resulting value of the same type in a variable *ÿwasÿ.*

SUB *ÿvarÿ ÿsymb1ÿ ÿsymb2ÿ* Subtraction of two numeric values
Subtracts *ÿsymb2ÿ* from *ÿsymb1ÿ* (must be of type int) and stores the resulting value of the same type in variable *ÿvarÿ.*

MUL *ÿvarÿ ÿsymb1ÿ ÿsymb2ÿ* Multiply two numeric values
Multiplies *ÿsymb1ÿ* and *ÿsymb2ÿ* (must be of type int) and stores the resulting value of the same type in variable *ÿvarÿ.*

IDIV *ÿvarÿ ÿsymb1ÿ ÿsymb2ÿ* Division of two integer values
Integer divides an integer value from *ÿsymb1ÿ* by another integer value from *ÿsymb2ÿ* (both must be of type int) and assigns the result of type int to the variable *ÿvarÿ.* Dividing by zero causes error 57.

LT/GT/EQ *ÿvarÿ ÿsymb1ÿ ÿsymb2ÿ* Relational operators less than, greater than, equal to
The instruction evaluates the relational operator between *ÿsymb1ÿ* and *ÿsymb2ÿ* (of the same type; int, bool or string) and writes a bool result (false if invalid or true if valid) into *ÿvarÿ* corresponding sessions). Strings are compared lexicographically and false is less than true. For calculating fuzzy inequalities can be AND/OR/NOT. With an operand of type nil (another source operand is of any type) can only be compared with the EQ instruction, otherwise error 53.

AND/OR/NOT *ÿvarÿ ÿsymb1ÿ ÿsymb2ÿ* Basic Boolean operators
Applies conjunction (logical AND)/disjunction (logical OR) to operands of type bool *ÿsymb1ÿ* and *ÿsymb2ÿ* or the negation of *ÿsymb1ÿ* (NOT has only 2 operands) and writes the bool result to *ÿwasÿ.*

INT2CHAR *ÿvarÿ ÿsymbÿ* The Converting an integer to a character
numeric value *ÿsymbÿ* is converted to a character that forms a single-character string according to Unicode assigned to *ÿvarÿ.* If *ÿsymbÿ* is not a valid ordinal character value in Unicode (see function mb_chr in PHP 8), error 58 will occur.

STRI2INT *ÿvarÿ ÿsymb1ÿ ÿsymb2ÿ* The ordinal value of the character
Stores in *ÿvarÿ* the ordinal value of the character (according to Unicode) in the string *ÿsymb1ÿ* at position *ÿsymb2ÿ* (indexed from zero). Indexing outside the given string results in error 58. See the mb_ord function in PHP 8.

5.4.4 Input-output instructions

READ *ÿvarÿ ÿtypeÿ* Read a value from standard input
Retrieves one value according to the specified type *ÿtypeÿ* ÿ {int, string, bool} and stores this value in variable *ÿvarÿ.* Load it using the ready-made Reader class of *the ipp-core framework.* When of an incorrect or missing input, the value nil@nil will be stored in the variable *ÿvarÿ .*

WRITE *ÿsymbÿ* Write a value to standard output
Prints the value of *ÿsymbÿ* to standard output. Except for type bool and value nil@nil is the format
listing compatible with the **print** command of the Python 3 language with the additional parameter end='' (to prevent
additional newlines). A truth value is written as true and false as
false. The value nil@nil is output as an empty string.

5.4.5 Working with strings

CONCAT *ÿvarÿ ÿsymb1ÿ ÿsymb2ÿ* Stores Concatenation of two strings
in variable *ÿvarÿ* the string resulting from the concatenation of two string operands *ÿsymb1ÿ* and
ÿsymb2ÿ (other types are not allowed).

STRLEN *ÿvarÿ ÿsymbÿ* Find the length of a string
Finds the number of characters (length) of the string in *ÿsymbÿ ,* and that length is stored as an integer in *ÿvarÿ.*

GETCHAR *ÿvarÿ ÿsymb1ÿ ÿsymb2ÿ* Return the character of the string
Stores in *ÿvarÿ* a string from one character in the string *ÿsymb1ÿ* at position *ÿsymb2ÿ* (indexed by the whole
number from zero). Indexing outside the given string results in error 58.

SETCHAR *ÿvarÿ ÿsymb1ÿ ÿsymb2ÿ* Modifies Change the string character
the character of the string stored in the variable *ÿvarÿ* at position *ÿsymb1ÿ* (integer-indexed
from zero) to a character in the string *ÿsymb2ÿ* (the first character if *ÿsymb2ÿ* contains multiple characters). Resultant
the string is again stored in *ÿvarÿ.* When indexing outside of the *ÿvarÿ* string or in the case of empty
string in *ÿsymb2ÿ* will result in error 58.

5.4.6 Working with types

TYPE *ÿvarÿ ÿsymbÿ* Find the type of the given symbol
Dynamically detects the symbol type *ÿsymbÿ* and writes a string indicating this type to *ÿvarÿ* (int, bool,
string or nil). If *ÿsymbÿ* is an uninitialized variable, it indicates its type with an empty string.

5.4.7 Program Flow Control Instructions

The non-terminal *ÿlabelÿ* indicates a label that is used to indicate a position in the IPPcode24 code. When
jumping to a non-existent label will result in error 52.

LABEL *ÿlabelÿ* Label definition
A special instruction that uses a *ÿlabelÿ* label to mark an important position in the code as a potential target
arbitrary jump instructions. Attempting to create two labels with the same name on different ones
error 52 in places of the program.

JUMP *ÿlabelÿ* Unconditional jump on the sign
Performs an unconditional jump to the specified label *ÿlabelÿ.*

JUMPIFEQ *ÿlabelÿ ÿsymb1ÿ ÿsymb2ÿ* Conditional label jump on equality
If *ÿsymb1ÿ* and *ÿsymb2ÿ* are of the same type or either operand is nil (otherwise error 53) and
at the same time, their values are equal, so it jumps to the label *ÿlabelÿ.*

JUMPIFNEQ *ÿlabelÿ ÿsymb1ÿ ÿsymb2ÿ* Conditional jump to label on inequality
If *ÿsymb1ÿ* and *ÿsymb2ÿ* are of the same type or any operand is nil (otherwise error 53), then
in case of different values, it jumps to the label *ÿlabelÿ.*

EXIT *ÿsymbÿ* Exit interpretation with return code
Terminates program execution, possibly prints statistics and terminates the interpreter with a return code
ÿsymbÿ, where *ÿsymbÿ* is an integer in the interval 0 to 9 (inclusive). Invalid integer value *ÿsymbÿ*
leads to error 57.

5.4.8 Debugging instructions

The following debug instructions (DPRINT and BREAK) must not affect standard output. We will not test their actual functionality, but they may appear in the tests.

DPRINT *ÿsymbÿ*	Print value to stderr Supposed to print the specified value *ÿsymbÿ* to standard error output (stderr).
BREAK	Write interpreter status to stderr It is supposed to write the status of the interpreter (eg position in the code, content of memory frames, number of executed instructions) to the standard error output (stderr) at a given moment (ie during the execution of this instruction).

Reference

[1] Extensible Markup Language (XML) 1.0. W3C. World Wide Web Consortium [online]. 5th edition. 26 November 2008 [cit. 2020-02-03]. Available from: https://www.w3.org/TR/xml/

[2] A7Soft JExamXML is a java based command line XML diff tool for comparing and merging XML documents. c2018 [cit. 2020-02-03]. Dostupné z: https://www.a7soft.com/jexamxml.html

[3] Kÿivka, Z., et al.: Assignment of a project from the subject IFJ and IAL. c2023 [cit. 2024-01-22]. Available from: https://www.fit.vutbr.cz/study/courses/IFJ/private/projekt/ifj2023.pdf

[4] The text/markdown Media Type. Internet Engineering Task Force (IETF). 2016 [cit. 2020-02-03]. Available from: https://tools.ietf.org/html/rfc7763

[5] Gamma, E., a kol.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[6] van Rossum, G., a kol.: PEP 8 – Style Guide for Python Code. c2023 [cit. 2024-01-30]. Available from: https://peps.python.org/pep-0008/

Assignment revision:

2024-02-14: Fixed typos. Opcode ordering specification for the --frequent statistic in the STATP extension. Finalizing the assignment.

2024-02-18: Fix function help for use with INT2CHAR and STRI2INT instructions.