Project Specification for IPPe 2022/2023

Zbyněk Křivka
E-mail: krivka@fit.vut.cz
MS Teams: xkrivk01@vutbr.cz

Simple Three-Address Code Parser and Interpreter

1 Basic characteristics

Implement two programs/scripts as a console applications implemented in Python 3.8[1]:

1. task: A parser (`parser.py`) checks lexical and syntactical correctness of a given source program in IPPeCode (text-based assembly-like language) and transforms the source code into a XML representation of the code.

2. task: An interpreter (`taci.py`) processes the given source program represented as three-address code (abbrv. 3-AC) instructions stored in XML format (as from the parser) and interprets it.

2 Organization

Teacher and consultant: Zbyněk Křivka, C229, krivka@fit.vut.cz, MS Teams: xkrivk01@vutbr.cz

During the lectures, there will be some tips for the project implementation, such as an introduction to Python 3, command-line argument parsing, XML format, and its parsing, and the discussion of a 3-AC interpreter architecture. If you need a personal consultation during the semester, send me an email. Please, fill "IPPe:" into the subject of every email sent to me!

Deadline Handover the implementation into StudIS before:

1. task: **March 28, 2023**, 23:59:59

2. task **April 25, 2023**, 23:59:59

May 3, 2023 Oral presentation + your slides (during the lecture), up to 2 points

Formal Requirements Both scripts will be a console application (i.e., no graphical user interface). They will not be interactive so they do not ask the user to input some additional information during its execution (apart from `taci.py`, which according to the interpreted reading instructions reads some input). The input data are given by script options through the command-line interface[2].

All error messages and auxiliary outputs not prescribed by the input program have to be written to the standard error output (stderr). The right kind of error has to be identified and the corresponding error code has to be returned according to Table 2 or 3. If the script works without problems or errors, the script returns zero (`0`) as its returning code[3].

For handover, each task will be packed in an archive consisting of the script (and maybe several auxiliary Python files) and the documentation in PDF. Name the archive by your login and the corresponding archive extension (e.g., xlogin99.zip). Only ZIP, RAR, TGZ or TBZ are supported.

[1] The scripts will be evaluated at `merlin.fit.vutbr.cz` server by `python3.8` which is a link to Python 3.8.13.
[2] See `argparse` library in Python 3.
[3] See `sys.exit()` function in Python 3.

Each archive with the solution can contain a folder named `tests`. This folder will contain at least ten different input files with programs that can be interpreted by your implementation of the interpreter, files with inputs that can be redirected to standard input and expected outputs from the standard output. Create these tests to demonstrate all implemented and interesting aspects of your project (especially if your implementation is somehow incomplete according to this specification).

Documentation The documentation of both scripts has to be written in English and be in PDF format. Please, use some English spell-checker and grammar checker. For `parser.py`, the documentation has to be at most one page long. For `taci.py`, the length should be shorter than two pages. Describe the design and implementation of the script (e.g., description of the most important data structures and algorithms). Give special attention to the problematic parts of your solution. Be clear, if you did not implement some specific feature from the specification. In the conclusion, evaluate your project and suggest some future development. The documentation shall not contain a copy of any part of the project specification (you can only refer to it).

3 IPPeCode: The Language Specification

IPPeCode is a simple non-structured non-typed programming language. IPPeCode program consists of instructions (or three-address codes; abbrv. 3-AC or TAC) and empty lines. Each instruction occupies exactly one line of code and consists of an operation code (case-insensitive[4] keyword) and few operands (case-sensitive). IPPeCode supports one-line comments that start with # (until the end of line). In the instruction, its operation code (abbrv. opcode) and operands are separated by one or more white characters (space or tabulator). Commonly, the operation code is a keyword describing the action/operation that is taken when the instruction is interpreted. Then, there are usually up to three operands: (1) the destination (abbrv. dst) describes where the result of the action will be stored; (2) the first (abbrv. src1) and (3) the second source operand of the operation (abbrv. src2). There are four kinds of operands: (1) variable, (2) constant integer literal, (3) constant string literal, and (4) label.

Consider _ (underscore), $ (dollar), & (ampersand), and % (percent) to be special characters. A variable identifier is defined as a non-empty sequence of letters, digits, and special characters starting with a letter or a special character, e.g. `$my_var`. A label is an identifier with the same rules as for variable identifier but always starts with @ (at), e.g. `@end`. An integer literal is a constant (decimal base) and consists of a sequence of digits that can be prefixed by a sign (e.g., `42`, `-6` or `+9`). The leading zeros cause a lexical error.

A string literal is a C-like string which is a sequence of printable characters (in UTF-8) surrounded by quotation marks and possibly containing escape sequences for the end of line (`\n`), backslash (`\\`)), and quotation mark (`\"`). For instance, `"My name is\n\"Zbynk Kivka\"."`

Variables in IPPeCode are dynamically typed, so the type of a variable is given dynamically by the value assigned to the variable. IPPeCode supports two types: integer and string. All variables in IPPeCode are global. A variable is defined and initialized by its occurrence as the destination (`dst`) of an instruction. The variable lifetime spans until the program ends. A variable may be reassigned (including the change of its type) by another occurrence as a destination of an instruction.

The complete list of 3-AC instructions is in Table 1 where x and y are variable names or literals, z is a variable name, and ℓ is a label. The start point of the whole program is the first 3-AC instruction.

[4]Case-insensitive means that upper-case and lower-case letters are understood as equal. For example, `ADD` and `Add` is the same keyword.

Opcode	dst	src1	src2	Description
`MOV`	z	x		Assign a value of x to z.
`ADD`	z	x	y	Assign the addition of x and y into z (integers).
`SUB`	z	x	y	Assign the subtraction of y from x into z (integers).
`MUL`	z	x	y	Assign the multiplication of x and y into z (integers).
`DIV`	z	x	y	Assign the division of x by y into z (integers).
`READINT`	z			Read an integer value from the standard input into z.
`READSTR`	z			Read a string from the standard input into z.
`PRINT`		x		Write the value of x to the standard output.
`LABEL`	ℓ			Possible target of a jump (call) instruction named ℓ.
`JUMP`	ℓ			Jump to label ℓ.
`JUMPIFEQ`	ℓ	x	y	Jump to label ℓ if $x = y$.
`JUMPIFLT`	ℓ	x	y	Jump to label ℓ if $x < y$.
`CALL`	ℓ			Save the program counter and jump to label ℓ.
`RETURN`				Load the last saved program counter and jump to it.
`PUSH`		x		Push a value of x to the data stack.
`POP`	z			Pop the top of the data stack and save it into z.

Table 1: The List of Syntax and Semantics of IPPeCode Instructions

During the interpretation there are some semantic and run-time checks to be performed to handle the error codes given in Table 3. More specifically, the source operands (`src1` and `src2`) of arithmetic instructions (`ADD`, `SUB`, `MUL`, `DIV`) and conditional jumps (`JUMPIFEQ`/`LT`) must be of the same type, otherwise error (no implicit conversions).

Note that `CALL` and `RETURN` instructions use call stack to store/load program counter.

Simple example of a program in IPPeCode (textual representation)

```
# Simple example: read a string from a user and write it 3-times.
MOV counter 0  #initialize variable with the empty string
READSTR Input
# iteration until the condition is true
LABEL @WHILE_
JUMPIFEQ @End counter +3
PRINT Input
PRINT "\n"
ADD     counter     counter     1
JUMP @WHILE_
LABEL @End
```

4 TASK 1: Parser Specification - `parser.py` (6+1 points)

Script `parser.py` will:

1. read the textual source code of IPPeCode (see the specification in Section 3) from the given file and checks its syntax;

Code	Description
11	Parsing Error: Unknown operation code of instruction.
12	Parsing Error: Missing or excessing operand of instruction.
14	Parsing Error: Bad kind of operand (e.g., label instead of variable).
17	Other lexical and syntax errors.
19	Internal errors.

Table 2: Return Codes (retcode) of the Parser

2. generate XML representation of the parsed source code into the given output file according to the specification bellow.

The script will get the parameters from the command-line:

`python3.8 parser.py` *`source`* `[`*`output`*`]`

- *`source`* input textual file with an IPPeCode source code (if just `-`, then read `stdin` instead of a physical file).
- *`output`* output file for the resulting XML representation in UTF-8 (if just `-`, then write into `stdout` instead of a physical file). If *output* exists, rewrite it; otherwise, create the file. Do not add any extension by default. If omitted, the default output file is `out.xml`.

4.1 XML representation of IPPeCode

After the mandatory XML header[5], the root element `program` (including mandatory text attribute `language` with value IPPeCode) follows. The root element consists of elements `tac` for individual instructions. Every `tac` element includes the mandatory `order` attribute that holds the order of the instruction (unbroken sequence starting with 1, no leading zeros) and the mandatory `opcode` attribute storing the value of operation code always written in upper case. In addition, `tac` element contains the elements for the operands/arguments of the instruction: `dst` for the destination operand, `src1` for the potential first source operand and `src2` for the second potential operand. The operand elements include mandatory attribute `type` with possible values: `integer`, `string`, `label`, and `variable` determining that the contained textual element represents integer literal, string literal (without the surrounding quotation marks), label (including @ at the beginning), or variable name, respectively.

There is no conversion of the integer values (do not throw away even plus sign) before their write up into XML. For string literal, leave escape sequences untouched, convert only colliding characters because of XML format (i.e., <, >, & using XML entities `<`, `>`, `&`). Resolve the problematic characters in identifiers as well.

A string literal is just a content of the corresponding XML element with attribute `type=string`. It can contain the end of line symbol (use `&eol;` entity) and few other entities in the XML input file (see Listing 1).

Simple example of a program in IPPeCode (XML representation)

[5]XML header including the version and encoding is `<?xml version="1.0" encoding="UTF-8" ?>`

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE program [
  <!ELEMENT program (tac+)>
  <!ELEMENT tac (dst?,src1?,src2?)>
  <!ELEMENT dst (#PCDATA)>
  <!ELEMENT src1 (#PCDATA)>
  <!ELEMENT src2 (#PCDATA)>
  <!ATTLIST program name CDATA #IMPLIED>
  <!ATTLIST tac opcode CDATA #REQUIRED>
  <!ATTLIST tac order CDATA #REQUIRED>
  <!ATTLIST dst type (integer|string|variable|label) #REQUIRED>
  <!ATTLIST src1 type (integer|string|variable) #REQUIRED>
  <!ATTLIST src2 type (integer|string|variable) #REQUIRED>
  <!ENTITY language "IPPeCode">
  <!ENTITY eol "&#xA;">
  <!ENTITY gt ">">
  <!ENTITY lt "<">
]>
<program name="Simple example of a program in IPPeCode">
  <tac opcode="MOV" order="1">
    <dst type="variable">counter</dst>
    <src1 type="integer">0</src1>
  </tac>
  <tac opcode="READSTR" order="2">
    <dst type="variable">Input</dst>
  </tac>
  <tac opcode="LABEL" order="3">
    <dst type="label">@WHILE_</dst>
  </tac>
  <tac opcode="JUMPIFEQ" order="4">
    <dst type="label">@End</dst>
    <src1 type="variable">counter</src1>
    <src2 type="integer">+3</src2>
  </tac>
  <tac opcode="PRINT" order="5">
    <src1 type="variable">Input</src1>
  </tac>
  <tac opcode="PRINT" order="6">
    <src1 type="string">&eol</src1>
  </tac>
  <tac opcode="ADD" order="7">
    <dst type="variable">counter</dst>
    <src1 type="variable">counter</src1>
    <src2 type="integer">1</src2>
  </tac>
  <tac opcode="JUMP" order="8">
    <dst type="label">@WHILE_</dst>
  </tac>
  <tac opcode="LABEL" order="9">
    <dst type="label">@End</dst>
  </tac>
</program>
```

5 TASK 2: Interpreter Specification - `taci.py` (10+1 points)

Script `taci.py` will:

Code	Description
20	Parsing Error during the parsing of XML, invalid XML input, file cannot be opened.
21	Semantic Error during the semantic checks (e.g., a label occurs several times).
23	Run-time Error: Jump/call to a non-existing label.
24	Run-time Error: Read access to a non-defined variable.
25	Run-time Error: Division by zero using `DIV` instruction.
26	Run-time Error: `READINT` get invalid value (not an integer).
27	Run-time Error: Operands of incompatible type.
28	Run-time Error: Pop from the empty (data/call) stack is forbidden[6].
30	Other run-time errors.
99	Internal errors.

Table 3: Return Codes (retcode) of the Interpreter

1. Parse the command-line option(s) to get the filename with the input program to interpret.
2. Open and parse the input file using an XML parser to get an internal representation (in the form of 3-AC) of the given source program.
3. Interpret the given program instruction by instruction according to their semantics (see Table 1).

The interpreter will be a console application that takes one mandatory command-line argument with the filename of the source program with the 3-AC instructions in XML format. The filename can be given also with relative or absolute path.

`python3.8 taci.py [--input=`*`infile`*`] `*`program`*` [`*`output`*`]`

- `--input=`*`infile`* input file with data for `READINT` and `READSTR` instructions in the program. Do not add any extension by default. If omitted, the default input is taken from `stdin`.
- *`program`* program file with XML representation of an IPPeCode source code (mandatory parameter).
- *`output`* output file for the texts output by `PRINT` instructions. Do not add any extension by default. If omitted, the output is written into standard output (`stdout`).

Listing 1: Document Type Definition (DTD) for XML representation

```
<!DOCTYPE program [
  <!ELEMENT program (tac+)>
  <!ELEMENT tac (dst?,src1?,src2?)>
  <!ELEMENT dst (#PCDATA)>
  <!ELEMENT src1 (#PCDATA)>
  <!ELEMENT src2 (#PCDATA)>
  <!ATTLIST program name CDATA #IMPLIED>
  <!ATTLIST tac opcode CDATA #REQUIRED>
  <!ATTLIST tac order CDATA #REQUIRED>
```

[6]For instance, RETURN while the call stack is empty leads to this error.

```
  <!ATTLIST dst type (integer|string|variable|label) #REQUIRED>
  <!ATTLIST src1 type (integer|string|variable) #REQUIRED>
  <!ATTLIST src2 type (integer|string|variable) #REQUIRED>
  <!ENTITY language "IPPeCode">
  <!ENTITY eol "&#xA;">
  <!ENTITY gt ">">
  <!ENTITY lt "<">
]>
```

The input XML file will be always well-formed and valid according to its document type definition (DTD) (see Listing 1). The presence of DTD from Listing 1 in the input XML file is optional.

References:

- Extensible Markup Language (XML) 1.0. W3C. World Wide Web Consortium [online]. 5th Edition. 2008-11-26. Available from `http://www.w3.org/TR/xml/`

6 Extensions

If you implement some of the extensions in your project, note it in the documentation.

The support of `string` variables (up to 3 bonus points)

- `CONCAT z x y` Assign the concatenation of x and y into z (strings).
- `GETAT dst src i` Assign the one-character string at (zero-based) index i of string *src* into *dst* (string).
- `LEN dst src` Assign the length of *src* (string) into *dst* (integer).
- `STRINT dst src` Convert a string *src* into an integer variable *dst*.
- `INTSTR dst src` Convert an integer *src* into a string variable *dst*.

If you have an idea for another extension, please, let me know and we will discuss it whether it is a good idea and whether you can get some extra points for it.