Documentation for TACI (Simple Three-Address Code Interpreter)

Marios Pafitis (xpafit00)

Code Execution

This is an interpreter implementation for three-address code instructions stored in XML format. The interpreter is implemented in Python 3. The program is a console application and you can run it by using the command:

python taci.py someXML.xml

Any input for the READINT and READSTR opcodes must be given in run-time, in the standard input. The interpreter is using standard error for the error messages. They following are code descriptions for each type of error.

Code Description:

3	Parsing Error during the parsing of XML, invalid XML input, file cannot be opened.
5	Semantic Error during the semantic checks (e.g. a label occurs several times).
10	Run-time Error: Jump to a non-existing label or call to non-existing function.
11	Run-time Error: Read access to non-defined or non-initialized variable.
12	Run-time Error: Division by zero using DIV instruction.
13	Run-time Error: READINT get invalid value (not an integer).
14	Run-time Error: Operands of incompatible type.
15	Run-time Error: Pop from the empty (data/call) stack is forbidden.
20	Other run-time errors.
99	Internal errors.

Functionality

The interpreter recognizes the following opcodes for the base functionalities and for the extra functionalities:

Base Functionalities:

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.
PRINT		x		Write the value of x to the standard output.
LABEL	l			Possible target of a jump (call) instruction named `.
JUMP	l			Jump to label l.
JUMPIFEQ	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

Extra Functionalities:

READSTR	str			Read a string from the standard input into str variable.
CONCAT	z	x	y	Assign the concatenation of x and y into z (strings).
GETAT	dst	src	i	Assign the one-character string at 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.

The parsing of XML is controlled by the argparse module and xml.etree.ElementTree. Also, the interpreter uses the sys module.

The implementation of the interpreter uses a different function for each opcode, and all the functions are managed by the controlOpCode() function.

The interpreter has a Data_Stack for store / load data by using the PUSH / POP opcode. The PC_Stack is used in the case of function calls in which it stores the return address. We use the CALL / RETURN opcode when we want to store / load the Program Counter (PC) before accessing and after leaving a function. We use the labels dictionary to store all the labels at the beginning of the execution. The same time we store the instructions into the instruction list. We pass through the instruction list to execute the instructions. If we want to read or store any variables, we use the assigned dictionary. There we can save data from different data types.

The opcodes MOV, ADD, SUB, MUL and DIV support only functions with integers. In case of the input is not n integer the functions try to cast it. If it throws an exception then we return an error code. For these opcodes you can use them either with variables or with literals. In the case of literals, you have to specify that the kind='literal' otherwise you will get an error.

For the READINT opcode if you don't give an integer as input, the method will return an error code after casting failure.

Conclusion

Through this project I have learned many new things. I had never got in touch with either Python or XML formats. I was impressed about the freedom and how easy is to learn Python. The TACT Three-Address Code Interpreter is implementing enough number of instructions for simple recursive programs. Maybe in the future I will implement some instructions for binary numbers such as AND, OR, XOR and SHIFT. All of the functionalities are tested and they work properly. It is important for the XML file to follows the specifications.

Specifications

For the destination, source1 and source2 use the <dst>, <src1> and <src2> tags respectively. For all the opcodes if you are going to use a literal you have to specify it, by saying kind='literal'. If you don't do that the interpreter is going to search the assigned dictionary for a variable such name and most probably is going to return an error code. If you want to use a variable as source for a specific instruction you don't have to specify it at all.

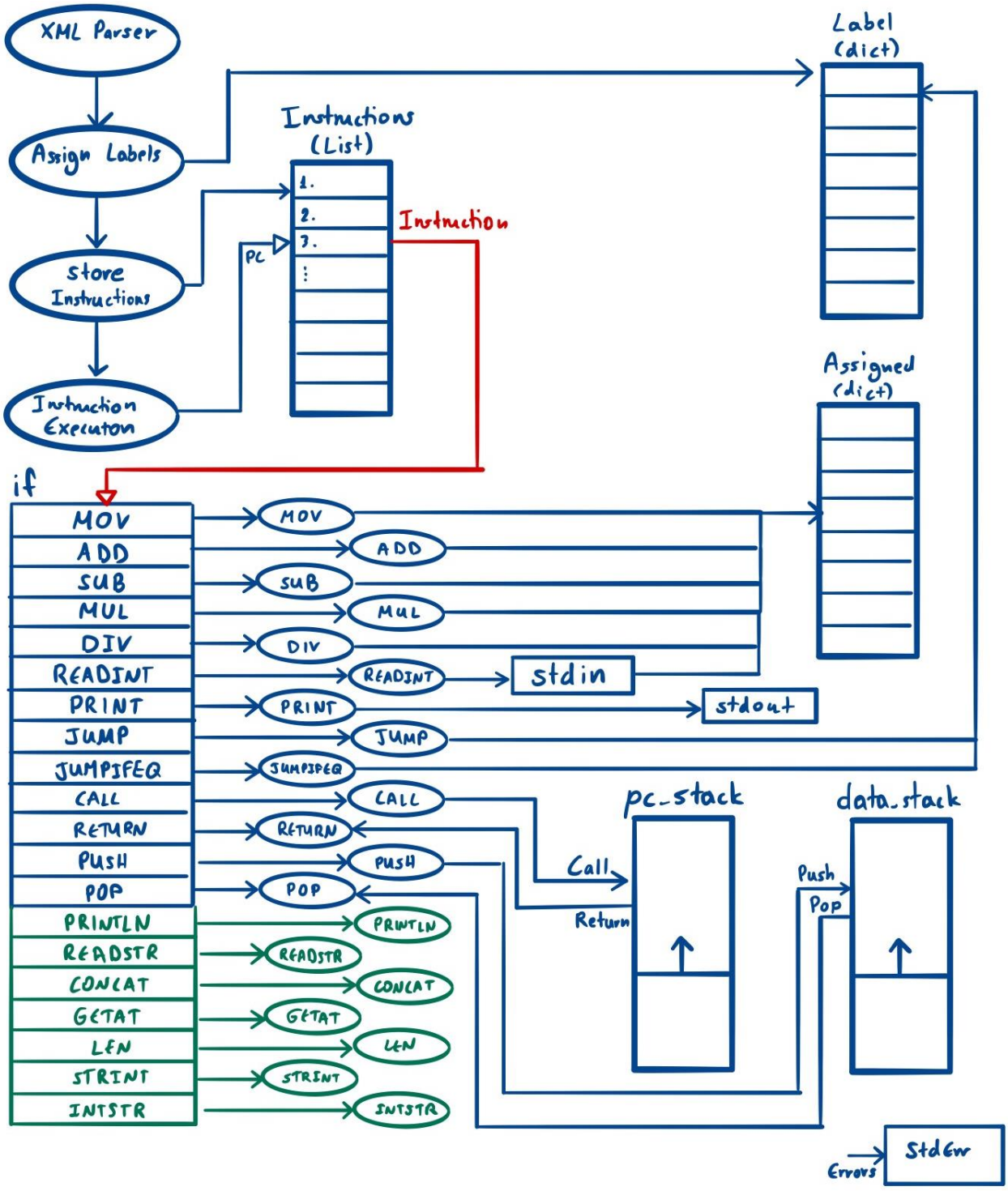
For Example:

Literal

```
<taci opcode="MOV">
    <dst>a</dst>
    <src1 kind="literal">20</src1>
  </taci>
```

Variable

```
<taci opcode="MOV">
    <dst>a</dst>
    <src1>b</src1>
  </taci>
```



TACI Three-Address Code Interpreter Schema