Lab 2
Interpreters and Types

Objective

- Understand visitors as a way to traverse a tree.
- Implement a typer and an interpreter as visitors.
- This is due **on TOMUSS (NO EMAIL PLEASE)**: deadline and info on the course's webpage.

EXERCISE #1 ▶ **Lab preparation**
Make sure you have read and understood REGLES-TPs.md on the course homepage.

In the lab's git repository (`mif08-2025`, if you don't have it already, see the beginning of lab1).

```
git commit -a #push is not allowed
git pull
```

will provide you all the necessary files for this lab in `TP02/` and `MiniC/`.

ANTLR4 and `pytest` should be installed and working like in previous lab, if not [1]:

```
python3 -m pip install --upgrade pytest pytest-cov pytest-xdist coverage iniconfig
```

2.1 Typing Annotations in Python

Python is originally a dynamically typed language, i.e. types are associated with values at runtime, but no check is done statically. Recent versions of Python, however, allow adding static typing annotations in the code. You don't *need* to add these annotations, but having them allows static typecheckers like Mypy or Pyright to find errors in the code before running it. In this course, we will use Pyright. You don't need to understand all the details of typing annotations, but you need to get familiar with them to understand the code provided to you.

If you work on your own machine, if needed, install pyright. You will also soon need to install RiscV tools, and we provide a pre-compiled archive containing it and Pyright (you may also use a Docker image, or a virtual machine), so the simplest is that you install all this now. Instructions on how to install all this on your machine is available in `INSTALL.md` on the course's webpage.

On the Nautibus machines, Pyright and the RiscV toolchain are installed, but you still have to add the following line to your `~/.bashrc` (create it if it doesn't exist):

```
source /home/tpetu/Enseignants/matthieu.moy/mif08/setup.sh
```

EXERCISE #2 ▶ **Basic type checking**
Consider the following code (available as `TP02/python-typing/typecheck.py` in your repository):

```python
# Typing annotations for variables:
# name: type
int_variable: int
float_variable: float
int_variable = 4.2 # Static typing error, but no runtime error
float_variable = 42.0 # OK
float_variable = int_variable # OK


# Typing annotations for functions (-> means "returns")
def int_to_string(i: int) -> str:
    return str(i)
```

[1] The `coverage` and `iniconfig` packages are not always needed but may solve compatibility issues with some versions of `pytest-cov`, yielding `pytest-cov: Failed to setup subprocess coverage` messages in some situations. The `--upgrade` option ensures we install the latest version even if an old version is already installed. Without it, the version compatibility issue would remain.

```
print(int_to_string('Hello')) # Static typing error, but no runtime error
print(int_to_string(42) / 5) # Both static and runtime error
```

Run the code in the Python interpreter:
`python3 typecheck.py`
Check for static typing errors using `pyright`:
`pyright typecheck.py`
Note how static typechecking finds more error in your code, and possibly saves you a lot of debbugging time.
Try modifying the code to get rid of typing errors.

EXERCISE #3 ▶ **Type unions**
Play with (execute, typecheck, read the comments) the following code (available as `TP02/python-typing/type_unions.py` in your repository):

```
from typing import List

# int | float means ``either an int or a float''.
NUMBER = int | float # or Union[int, float] with Union imported from typing


def add_numbers(a: NUMBER, b: NUMBER) -> NUMBER:
    return a + b


# Both int and floats can be passed to the function
print(add_numbers(1, 4.3))


def divide_numbers(a: NUMBER, b: NUMBER) -> float:
    return a / b


print(divide_numbers(1, 2))

# Declare the type of a list whose elements are numbers.
LIST_OF_NUMBERS = List[NUMBER]


def increment(a: LIST_OF_NUMBERS) -> LIST_OF_NUMBERS:
    return [x + 1 for x in a]


print(increment([1, 2, 3]))

# Skip the end if you are late.

# The type DEEP_LIST_OF_NUMBERS is a special case since it references itself.
# The identifier DEEP_LIST_OF_NUMBERS cannot be used before the end of its
# initialization, but the circular dependency can be broken using the string
# 'DEEP_LIST_OF_NUMBERS' instead.
DEEP_LIST_OF_NUMBERS = NUMBER | List['DEEP_LIST_OF_NUMBERS']


def deep_increment(d: DEEP_LIST_OF_NUMBERS) -> DEEP_LIST_OF_NUMBERS:
    if isinstance(d, list):
```

```
        # Note the unusual typing rule applied by Pyright here: because we are
        # in the 'isinstance(d, list)' branch, it knows that d is a list,
        # and accepts to iterate over it.
        return [deep_increment(e) for e in d]
    else:
        # ... and here, in the 'else' branch Pyright knows that d is
        # not a list,
        # and can deduce that it is a NUMBER.
        return d + 1


print(deep_increment([1, [2, 3]]))
```

The syntax `int | float` (or equivalently `Union[int, float]`) means "either an `int` or a `float`"; `List[NUMBER]` means "a list whose elements are numbers".

2.2 Demo: Implicit tree walking using Visitors

2.2.1 Interpret (evaluate) arithmetic expressions with visitors

In the previous lab, we used an "attribute grammar" to evaluate arithmetic expressions during parsing. Today, we are going to let ANTLR build the syntax tree entirely, and then traverse this tree using the *Visitor* design pattern[2]. A visitor is a way to separate algorithms from the data structure they apply to.

For every possible type of node in your AST, a visitor will implement a method that will apply to nodes of this type.

EXERCISE #4 ▶ **Demo: arithmetic expression interpreter (`TP02/arith-visitor/`)**
Observe and play with the `Arit.g4` grammar and its PYTHON Visitor on `myexample` :
`$ make && make ex`
Note that unlike the "attribute grammar" version that we used previously, the `.g4` file does not contain Python code at all.

Have a look at the `AritVisitor.py`, which is automatically generated by ANTLR4: it provides an abstract visitor whose methods do nothing except a recursive call on children. Have a look at the `MyAritVisitor.py` file, observe how we override the methods to implement the interpreter, and try using `print` instructions to observe how the visitor actually works.

Also note the `#blabla` pragmas after each rules in the g4 file. They are here to provide ANTLR4 a name for each alternative in grammar rules. These names are used in the visitor classes, as method names that get called when the associated rule is found (eg. `#foo` will get `visitFoo(ctx)` to be called).

We depict the relationship between visitors' classes in Figure 2.1.

[2] `https://en.wikipedia.org/wiki/Visitor_pattern`

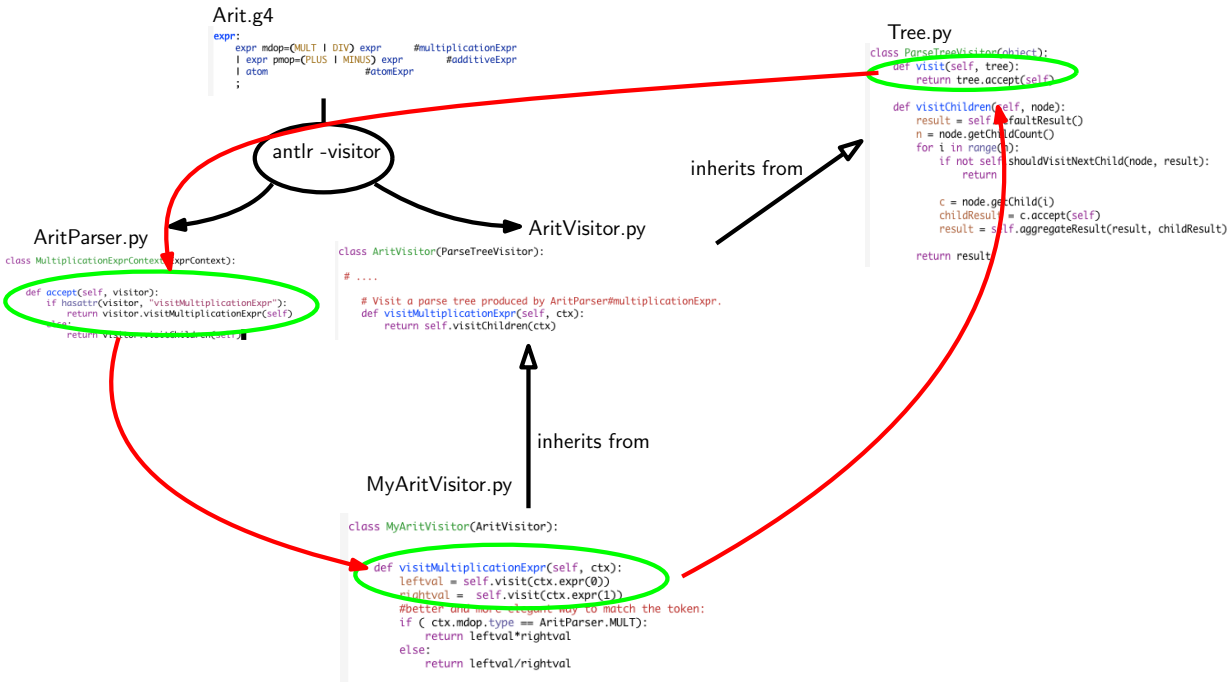

Figure 2.1: Visitor implementation Python/ANTLR4. ANTLR4 generates `AritParser` as well as `AritVisitor`. This `AritVisitor` inherits from the ParseTree visitor class (defined in `Tree.py` of the ANTLR4-Python library, use `find` to search for it). When visiting a grammar object, a call to `visit` calls the highest level `visit`, which itself calls the accept method of the Parser object of the good type (in `AritParser`) which finally calls your implementation of `MyAritVisitor` that match this particular type (here Multiplication). This process is depicted by the red cycle.

2.2.2 Basic rules to write an ANTLR4 visitor

what you can write in Python code is dictated by the `.g4` file:

- For each alternative of each rule labeled `#fooBar`, write a method `visitFooBar(self, ctx)` where `ctx` is the corresponding parse subtree. Note the case change, `#fooBar` has lower-case `f` but `visitFooBar` has upper-case `F`.
- For each element `elem` in the right-hand side of the rule (lower-case non-terminal or upper-case terminal), you can access its value with `ctx.elem()`. For non-terminal (lower-case) elements, the corresponding value is a tree. For terminal (upper-case) elements, it is a token of type `antlr4.Token.CommonToken`. Tokens have in particular a field `type` whose value is `AritParser.`*token-name*.
- When the element appears several times, access the n-th instance using `ctx.elem(`n`)` (starting with 0). Note that when there's only one instance, you cannot use `ctx.elem(0)` but can only write `ctx.elem()`.
- When the element is named like `left=expr` in the rule, access it with `ctx.left` (no parenthesis this time).
- Both trees and tokens have a `getText()` method returning the corresponding text in the source code.
- Recursive calls on sub-trees are written as `self.visit(subtree)`.

Example: when a ANTLR4 rule contains an operator alternative such as:

```
| expr addop=(PLUS | MINUS) expr #additiveExpr
```

you can use the following code in your implementation of `visitAdditiveExpr` to match the operator:

```
    def visitAdditiveExpr(self, ctx):
        leftval = self.visit(ctx.expr(0))
        rightval = self.visit(ctx.expr(1))
        if ctx.addop.type == AritParser.PLUS:
            return leftval + rightval
        else:
            return leftval -rightval
```

Note that we wrote `PLUS` and `MINUS` in the same rule to have the same level of precedence, and avoid the issues we had in previous lab.

2.2.3 Application to MiniC Language

The objective is now to use visitors, to type and interpret MiniC programs, whose syntax is depicted in Figure 2.2.

```
grammar MiniC;

prog: function* EOF #progRule;

// For now, we don't have "real" functions, just the main() function
// that is the main program, with a hardcoded profile and final
// 'return 0' (actually a 'return INT' because we don't have a ZERO
// lexical token).
function: INTTYPE ID OPAR CPAR OBRACE vardecl_l block
       RETURN INT SCOL CBRACE #funcDef;

vardecl_l: vardecl* #varDeclList;

vardecl: typee id_l SCOL #varDecl;


id_l: ID #idListBase
    | ID COM id_l #idList
    ;

block: stat* #statList;

stat: assignment SCOL
    | if_stat
    | while_stat
    | print_stat
    ;

assignment: ID ASSIGN expr #assignStat;

if_stat: IF OPAR expr CPAR then_block=stat_block
       (ELSE else_block=stat_block)? #ifStat;

stat_block: OBRACE block CBRACE
         | stat
         ;

while_stat: WHILE OPAR expr CPAR body=stat_block #whileStat;


print_stat
    : PRINTLN_INT OPAR expr CPAR SCOL #printlnintStat
    | PRINTLN_FLOAT OPAR expr CPAR SCOL #printlnfloatStat
    | PRINTLN_BOOL OPAR expr CPAR SCOL #printlnboolStat
    | PRINTLN_STRING OPAR expr CPAR SCOL #printlnstringStat
    ;
```

Figure 2.2: MiniC syntax. We omitted here the subgrammar for expressions

EXERCISE #5 ▶ **Be prepared!**

In the directory `MiniC/` (outside `TP02/`), you will find:

- The MiniC grammar (`MiniC.g4`). Run `make` to run ANTLR on it and generate the corresponding Python files.
- Our "main" program (`MiniCC.py`) which does the parsing of the input file, then launches the Typing visitor, and if the file is well typed, launches the Interpreter visitor. In this lab it supports four modes:
  - `python3 MiniCC.py --mode parse <file>` checks whether the given file is syntactically valid MiniC code.
  - `python3 MiniCC.py --mode typecheck <file>` parses the given file and typechecks it.
  - `python3 MiniCC.py --mode eval <file>` parses, typechecks, and interprets the given program.

  Try it on some provided examples (e.g. in `TypingAndInterpret/tests/provided/examples-types/`), see what happens for well-typed and ill-typed programs (usually named `bad_*.c`).
- A directory `TypingAndInterpret/` where files relevant to this lab are.
- One complete visitor: `TypingAndInterpret/MiniCTypingVisitor.py`, and one to be completed: `TypingAndInterpret/MiniCInterpretVisitor.py`.
- Some test cases (`TypingAndInterpret/tests`), and a test infrastructure.

2.3 Typing the MiniC-language (MiniC/)

2.3.1 Informal Typing Specification for the MiniC Language

MiniC is a subset of C with stricter rules, and predefined aliases:

```
typedef char * string;
typedef int bool;
static const int true = 1;
static const int false = 0;
```

The informal typing rules for the MiniC language are:

- Variables must be declared before being used, and can be declared only once ;
- Binary operations (+, -, *, ==, !=, <=, &&, ||, ...) require both arguments to be of the same type (e.g. `1 + 2.0` is rejected) ;
- Boolean and integers are incompatible types (e.g. `while(1)` is rejected) ;
- Binary arithmetic operators return the same type as their operands (e.g. `2. + 3.` is a float, `1 / 2` is the integer division) ;
- Modulo (%) is accepted only on integers, not floats.
- + is accepted on string (it is the concatenation operator), no other arithmetic operator is allowed for string ;
- Comparison operators (==, <=, ...) and logic operators (&&, ||) return a Boolean ;
- == and != accept any type as operands ;
- Other comparison operators (<, >=, ...) accept int and float operands only.

We do not consider real functions, so all your test cases will contain only a `main` function, without argument and returning 0.

EXERCISE #6 ► **Demo: play with the Typing visitor**

We provide you the code of the typer for the MiniC-language, whose objective is to implement the typing rules of the course. Open and observe `TypingAndInterpret/MiniCTypingVisitor.py`, and predict its behavior on the following MiniC file:

```
int x;
x="blablabla";
```

Then, test with:

```
python3 MiniCC.py --mode eval TypingAndInterpret/tests/provided/examples-types/bad_type00.c
```

Observe the behavior of the visitor on all test files in `example-types/`.

- How do we handle a multiplication between `int` and `string` operands?
- How do we handle an assignment between a variable and an expression with the "wrong" type?
- How to we remember the declared type for each variable?

EXERCISE #7 ► **Demo: test infrastructure for incorrectly typed programs**

On incorrectly typed programs, what we expect from a good test infrastructure is that is capable of checking if we handled properly the case. This is solved by augmenting the pragma syntax of the previous lab. For instance:

```
int x;
x="blablabla";
// EXPECTED
// In function main: Line 5 col 2: type mismatch for x: integer and string
// EXITCODE 2
```

will be a successful test case. Any error (typing or runtime) must raise the exit code different from 0 (details below).

To see how the tests work, type:

```
make test-interpret
```

Obviously, many tests will fail, since you didn't write the code yet. If you get an error about the `--cov` argument, you didn't properly install `pytest-cov`. To allow compiling your MiniC programs with a regular C compiler, a `printlib.h` file is provided, and should be `#included` in all your MiniC test cases[3].

[3] Note that unlike real C, # is a comment in MiniC to avoid actually having to deal with `#include`

You will later add your own tests: add them all in the `TypingAndInterpret/tests/students/` directory (mandatorily).

Note that the test infrastructure does two things on each file:

- Run your interpreter, and check that its behavior matches the annotation in the source file, to find bugs in your interpreter. This corresponds to the lines `test_eval` in the output of `make test-interpret`.
- Run the same file through GCC and execute it, to check that the annotations in the source file are correct with a GCC execution, to find bugs in your annotations. This corresponds to the lines `test_expect` in the output. You can disable this check when GCC and your interpreter have different behavior (e.g. for division by zero), by adding the annotation `// SKIP TEST EXPECTED` in the source file. It is normal to have some `test_expect` skipped (you just don't have the check that your annotations are OK when this happens), but you should have all `test_eval` pass.

2.4 An interpreter for the MiniC-language

2.4.1 Informal Specifications of the MiniC Language Semantics

MiniC is a small imperative language inspired from C, with more restrictive typing and semantic rules. Some constructs have an undefined behavior in C and well defined semantics in MiniC:

- Variables that are not explicitly initialized in the program are automatically initialized:
  - to `0` for `int`,
  - to `0.0` for `float`,
  - to `false` for `bool`,
  - to the empty string `""` for `string`.
- Divisions and modulo by 0 must print the message "Division by 0" and stop program execution with status 1 (use `raise MiniCRuntimeError("Division by 0")` to achieve this in the interpreter).
- Conventions for division and modulo are the same as in C: division is truncated toward zero, and modulo is such that $(a/b) * b + a\%b = a$.

$$\begin{array}{rclrcl} 4/3 & = & 1 & 4\%3 & = & 1 \\ (-4)/3 & = & -1 & (-4)\%3 & = & -1 \\ 4/(-3) & = & -1 & 4\%(-3) & = & 1 \\ (-4)/(-3) & = & 1 & (-4)\%(-3) & = & -1 \end{array}$$

2.4.2 Implementation of the Interpreter

The semantics of the MiniC language (how to evaluate a given MiniC program) is defined by induction on the syntax. You already saw how to evaluate a given expression, this is depicted in Figure 2.3.

EXERCISE #8 ▶ **Interpreter rules (on paper)**
First fill the empty cells in Figure 2.4, then ask your teaching assistant to correct them.

EXERCISE #9 ▶ **Interpreter**
Now you have to implement the interpreter of the MiniC language. We give you the structure of the code and the implementation for numerical expressions and boolean expressions (`except modulo!`). You can reason in terms of "well-typed programs", since badly typed programs should have been rejected earlier.

Run the command:

```
make
python3 MiniCC.py --mode eval TypingAndInterpret/tests/provided/examples/test_print_int.c
```

and the interpreter will be run on `test_print_int.c`. **On the particular example `test_print_int.c` observe how integer values are printed.**

You still have to implement (in `MiniCInterpretVisitor.py`):

1. The modulo version of Multiplicative expressions (for the C language semantics of modulo).

literal constant c	`return int(c) or float(c)`
variable name x	`find value of x in dictionary and return it`

e_1+e_2	`v1 <- e1.visit()`
	`v2 <- e2.visit()`
	`return v1+v2`

true	`return true`
$e_1 < e_2$	`return e1.visit()<e2.visit()`

Figure 2.3: Interpretation (Evaluation) of expressions

x := e	`v <- e.visit()` `store(x,v) #update the value in dict`
println_int(e)	`v <- e.visit()` `print(v) # python's print`
S1; S2	`s1.visit()` `s2.visit()`
`if` *b* `then` *S*1 `else` *S*2	
`while` *b* `do` *S* `done`	

Figure 2.4: Interpretation for Statements (pseudo-code)

2. Variable declarations (`varDecl`) and variable use (`idAtom`): your interpreter should use a table (*dict* in PYTHON) to store variable definitions and check if variables are correctly defined and initialized. **Do**

not forget to initialize dict with the initial values (`0`, `0.0`, `False` or `""` depending on the type) for all variable declarations.

3. Statements: assignments, conditional blocks, tests, loops.

Error codes The exit code of the interpreter should be:

- 1 in case of runtime error (e.g. division by 0, absence of main function)
- 2 in case of typing error
- 3 in case of syntax error
- 4 in case of internal error (i.e. error that should never happen except during debugging)
- 5 in case of unsupported construct (should not be used in this lab, but you will need it to reject strings and floats during code generation)
- And obviously, 0 if the program is typechecked and executed without errors.

The file `MiniCC.py` in the skeleton already does this for you if you raise the right exception (see `Lib/Errors.py`). You need to use these codes as test annotations in programs raising errors:

- Programs raising a runtime error should be annotated with `// EXECCODE 1`
- Programs rejected by the interpreter before execution should be annotated with `// EXITCODE` *n*, with *n* being 2, 3, 4 or 5 as documented above.

The distinction between `EXECCODE` and `EXITCODE` seems subtle for an interpreter, but will be more obvious for a compiler, where `EXITCODE` will refer to the exit code of the *compiler*, and `EXECCODE` to the exit code of the *program's execution*.

EXERCISE #10 ▶ **Automated tests**

Test with `make test-interpret` **and write an appropriate test-suite**. You need not test the typechecker (i.e. tests raising a typing error are not needed). You must provide your own tests: they will be graded depending on their quality. The only outputs are the one from the `println_*` function or the following error messages: "*m* `has no value yet!`" (or possibly "`Undefined variable` *m*", but this error should never happen if your typechecker did its job properly) where *m* is the name of the variable. In case the program has no `main` function, the typechecker accepts the program, but it cannot be executed, hence the interpreter raises a "`No main function in file`" runtime error.

Test Infrastructure Tests work mostly as in the previous lab. By default, the testsuite is ran on all `.c` files in the `TypingAndInterpret/tests/` directory. You may restrict to a set of files using "`make test-interpret FILTER=...`". `FILTER` is either a single file or an extended wildcard like `FILTER=TypingAndInterpret/tests/provided/**/*.c` (`**` matches any directory hierarchy).

Source files should contain `// EXPECTED` and `// EXITCODE` *n* pragmas to specify the expected behavior of the compiler. They are special comments (the `//` is needed to keep compatibility with C, only the testsuite considers them as special). The `EXITCODE` corresponds to the exit codes described in Section 2.4.2.

For instance, if you fail `test_print_int.c` because you printed `43` instead of `42`, using the command `make test-interpret FILTER=TypingAndInterpret/tests/provided/examples/test_print_int.c` you will get this error:

```
____________TestInterpret.test_eval[TypingAndInterpret/tests/provided/examples/test_print_int.c] ____________

self = <test_interpreter.TestInterpret object at 0x7f05d6f86a40>,
filename = 'TypingAndInterpret/tests/provided/examples/test_print_int.c'

    @pytest.mark.parametrize('filename', ALL_FILES)
    def test_eval(self, filename):
        cat(filename)  # For diagnosis
        expect = self.get_expect(filename)
        eval = self.evaluate(filename)
        if expect:
>           self.assert_equal(eval, expect)
```

```
test_interpreter.py:48:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
self = <test_interpreter.TestInterpret object at 0x7f05d6f86a40>,
actual = testinfo(exitcode=0, execcode=0, output='43\n', linkargs=[], skip_test_expected=False)
expected = testinfo(exitcode=0, execcode=0, output='42\n', linkargs=[], skip_test_expected=False)

    def assert_equal(self, actual, expected):
        if expected.output is not None and actual.output is not None:
>           assert actual.output == expected.output, \
                "Output of the program is incorrect."
E           AssertionError: Output of the program is incorrect.
E           assert '43\n' == '42\n'
E             - 42
E             + 43

test_interpreter.py:35: AssertionError
```

And if you did not print anything at all when 42 was expected, the last lines would be this instead:

```
    def assert_equal(self, actual, expected):
        if expected.output is not None and actual.output is not None:
>           assert actual.output == expected.output, \
                "Output of the program is incorrect."
E           AssertionError: Output of the program is incorrect.
E           assert '' == '42\n'
E             - 42

test_interpreter.py:35: AssertionError
```

2.5 Final delivery

We recall that your work is **personal** and code copy (including tests) is **strictly forbidden**. Read REGLES-TPs.md again if in needed.

EXERCISE #11 ▶ **Archive**

The interpreter (all exercises in Section 2.4, i.e. Python code and tests, in `students/`) is due on TOMUSS (deadline on the course webpage). Type `make tar` to obtain the archive to send. Your archive must also contain a (minimal) `README-interpreter.md` with your name, your design choices, known bugs, if any. Don't spend more than 10 minutes on your `README-interpreter.md`.