Lab 3
Interpreters and Types

Objective

- Understand visitors.
- Implement typers, interpreters as visitors.
- This is due **on TOMUSS (NO EMAIL PLEASE)**: deadline and info on the course's webpage.

EXERCISE #1 ► **Lab preparation**
In the `mif08-labs20` directory: `git pull` will provide you all the necessary files for this lab in `TP03`. ANTLR4 and `pytest` should be installed and working like in Lab 2, if not :

```
python3 -m pip install --user pytest
```

The testsuite also uses `pytest-cov`, to be installed with[1]:

```
python3 -m pip install --user pytest-cov
python3 -m pip install --user --upgrade coverage
```

3.1 Demo: Implicit tree walking using Visitors

3.1.1 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 seperate algorithms from the data structure they apply to. For every possible type of node in your AST, a visitor will implement a function that will apply to nodes of this type.

EXERCISE #2 ► **Demo: arithmetic expression interpreter (`arith-visitor/`)**
Observe and play with the `Arit.g4` grammar and its PYTHON Visitor :

```
$ make ; make run < myexample
```

Note that unlike the "attribute grammar" version that we used previously, the `.g4` file does not contain Python code at all.

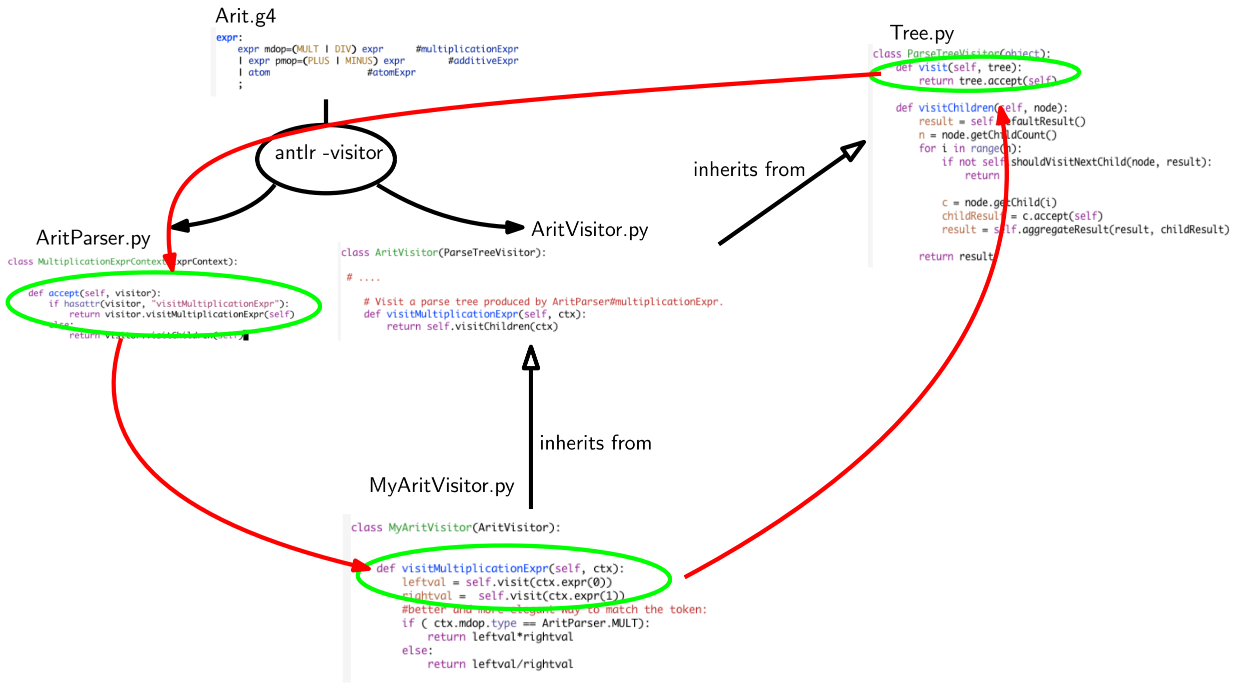
Have a look at the `AritVisitor.py`, which is automatically generated by ANTLR4: it provides an abstract visitor whose methods do nothing except a recursive call on children. Have a look at the `MyAritVisitor.py` file, observe how we override the methods to implement the interpret, and use `print` instructions to observe how the visitor actually work (print some node contents).

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 3.1.

[1] The second line is not always needed but may solve compatibility issues between versions of pytest-cov and coverage, yielding `pytest-cov: Failed to setup subprocess coverage` messages in some situations.

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



```
expr:
    expr mdop=(MULT | DIV) expr      #multiplicationExpr
    | expr pmop=(PLUS | MINUS) expr       #additiveExpr
    | atom                    #atomExpr
    ;
```

```
class MultiplicationExprContext(ExprContext):

    def accept(self, visitor):
        if hasattr(visitor, "visitMultiplicationExpr"):
            return visitor.visitMultiplicationExpr(self)
        else:
            return visitor.visitChildren(self)
```

```
class AritVisitor(ParseTreeVisitor):

 # ....

    # Visit a parse tree produced by AritParser#multiplicationExpr.
    def visitMultiplicationExpr(self, ctx):
        return self.visitChildren(ctx)
```

```
class ParseTreeVisitor(object):
    def visit(self, tree):
        return tree.accept(self)

    def visitChildren(self, node):
        result = self.defaultResult()
        n = node.getChildCount()
        for i in range(n):
            if not self.shouldVisitNextChild(node, result):
                return

            c = node.getChild(i)
            childResult = c.accept(self)
            result = self.aggregateResult(result, childResult)

        return result
```

```
class MyAritVisitor(AritVisitor):

  def visitMultiplicationExpr(self, ctx):
      leftval = self.visit(ctx.expr(0))
      rightval =  self.visit(ctx.expr(1))
      #better and more elegant way to match the token:
      if ( ctx.mdop.type == AritParser.MULT):
          return leftval*rightval
      else:
          return leftval/rightval
```

Figure 3.1: 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.

A last remark: when a ANTLR4 rule contains an operator alternative such as:

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

you can use the following code to match the operator:

```
if ( ctx.pmop.type == AritParser.PLUS):
    ...
```

The objective is now to use visitors, to type and interpret MiniC programs, whose syntax is depicted in Figure 3.2.

```
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'.
function: INTTYPE ID OPAR CPAR OBRACE vardecl_l block
       RETURN INT SCOL CBRACE #funcDecl;

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 stat_block #whileStat;


print_stat
    : PRINTINT OPAR expr CPAR SCOL #printintStat
    | PRINTFLOAT OPAR expr CPAR SCOL #printfloatStat
```

Figure 3.2: MiniC syntax. We omitted here the subgrammar for expressions

EXERCISE #3 ► **Be prepared!**

In the directory `MiniC/` (outside `TP03/`), you will find:

- The MiniC grammar (`MiniC.g4`).
- Our "main" program (`MiniCInterpreter.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.
- One complete visitor: `TP03/MiniCTypingVisitor.py`, and one to be completed: `TP03/MiniCInterpretVisitor.py`.
- Some test cases, and a test infrastructure.

3.2 Typing the MiniC-language (`MiniC/`)

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) ;
- + 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.

For now, we do not consider real functions, so all your test cases will contain only a `main` function, without argument and returning an integer. We will extend your code and write test-cases with several function definitions and calls in a further lab.

EXERCISE #4 ► **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 `MiniCTypingVisitor.py`, and predict its behavior on the following MiniC file:

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

Then, test with:

```
python3 MiniCInterpreter.py TP03/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 assignement between a variable and an expression with the “wrong” type?
- How to we remember the declared type for each variable?

EXERCISE #5 ► **Demo: test infrastructure for bad-typed programs**

On bad typed programs, what we expect from a good test infrastructure is that is 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). Now, type:

```
make tests
```

and observe (Typing tests are those concerning files in `examples-types/`). 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 `#include`d in all your MiniC test cases.

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

3.3 An interpreter for the MiniC-language

3.3.1 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).

3.3.2 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 3.3.

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	`let v1 = e1.visit() and v2 = e2.visit() in`
	`    return v1+v2`

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

Figure 3.3: Interpretation (Evaluation) of expressions

EXERCISE #6 ▶ **Interpreter rules (on paper)**
First fill the empty cells in Figure 3.4, then ask your teaching assistant to correct them.

EXERCISE #7 ▶ **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.

Type:
```
make
python3 MiniCInterpreter.py TP03/tests/provided/examples/test00.c
```
and the interpreter will be run on `test00.c`. **On the particular example `test00.c` observe how integer values, are printed.**

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

1. The modulo version of Multiplicative expressions.
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.** Refer to the three test files `bad_defxx.c` for the expected error messages.
3. Statements: assignments, conditional blocks, tests, loops.

x := e	`let v = e.visit() in`
	`   store(x,v) #update the value in dict`

println_int(e)	`let v = e.visit() in`
	`   print(v) #python print`

S1; S2	`s1.visit()`
	`s2.visit()`

`if` b `then` $S1$ `else` $S2$	
`while` b `do` S `done`	

Figure 3.4: Interpretation for Statements

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 lab3, but you will need it for strings and floats during code generation)
- And obviously, 0 if the program is typechecked and executed without error.

EXERCISE #8 ► **Automated tests**

Test with `make tests` **and appropriate test-suite**. If you get an error about the `--cov` argument, you didn't properly install pytest-cov. You must provide your own tests. 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`" error. To properly test the `bad_def*` files, you will have to edit the python test script `test_interpreter.py`[3]

[3] the linux command `find` is your friend: `find TP03/tests -iname "bad*"`

Test Infrastructure Tests work mostly as in the previous lab, with `// EXPECTED` and `// EXITCODE` *n* pragmas in the tests. They are special comments (the `//` is needed to keep compatibility with C, only the testsuite considers it as special). The `EXITCODE` corresponds to the exit codes described in Section 3.3.2.

For instance, if you fail `test00.c` because you printed 43 instead of 42, you will get this error:

```
_____________ TestCodeGen.test_expect[/path/test00.c] _____________

self = <test_interpreter.TestCodeGen object at 0x7f0e0aa369b0>
filename = '/path/to/test00.c'

    @pytest.mark.parametrize('filename', ALL_FILES)
    def test_expect(self, filename):
        expect = self.extract_expect(filename)
        eval = self.evaluate(filename)
        if expect:
>           assert(expect == eval)
E           assert '43\n1\n' == '42\n1\n'
E             - 43
E             + 42
E               1

test_interpreter.py:59: AssertionError
```

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

```
        if expect:
>           assert(expect == eval)
E           assert '42\n1\n' == '1\n'
E             - 42
E               1

test_interpreter.py:59: AssertionError
```

3.4 Final delivery

We recall that your work is **personal** and code copy is **strictly forbidden**.

EXERCISE #9 ▶ **Archive**

The interpreter (all exercises in Section 3.3) is due on TOMUSS (deadline on the course webpage).Type `make tar` to obtain the archive to send. Your archive must also contain tests (Tests must be in `students/`) and a (minimal) `README-interpreter.md` with your name, the functionality of the code, how to use it, your design choices, and known bugs, tests.