Lab 3
Interpreters and Types

Objective

- Understand visitors.
- Implement typers, interpreters as visitors.

The evaluator will be graded. Instructions and deadline on Chamilo too.

EXERCISE #1 ▶ **Lab preparation**
`git pull` will provide you all the necessary files for this lab in `TP03` and `MiniC`. ANTLR4 and `pytest` should be installed and working like in Lab 1, if not [1]:

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

3.1 Demo: Implicit tree walking using Visitors

This is a demo and a startup. Nothing has to be put on Chamilo for this section

3.1.1 Interpret (evaluate) arithmetic expressions with visitors

In the first 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 (`TP03/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 use `print` instructions to observe how the visitor actually works (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`

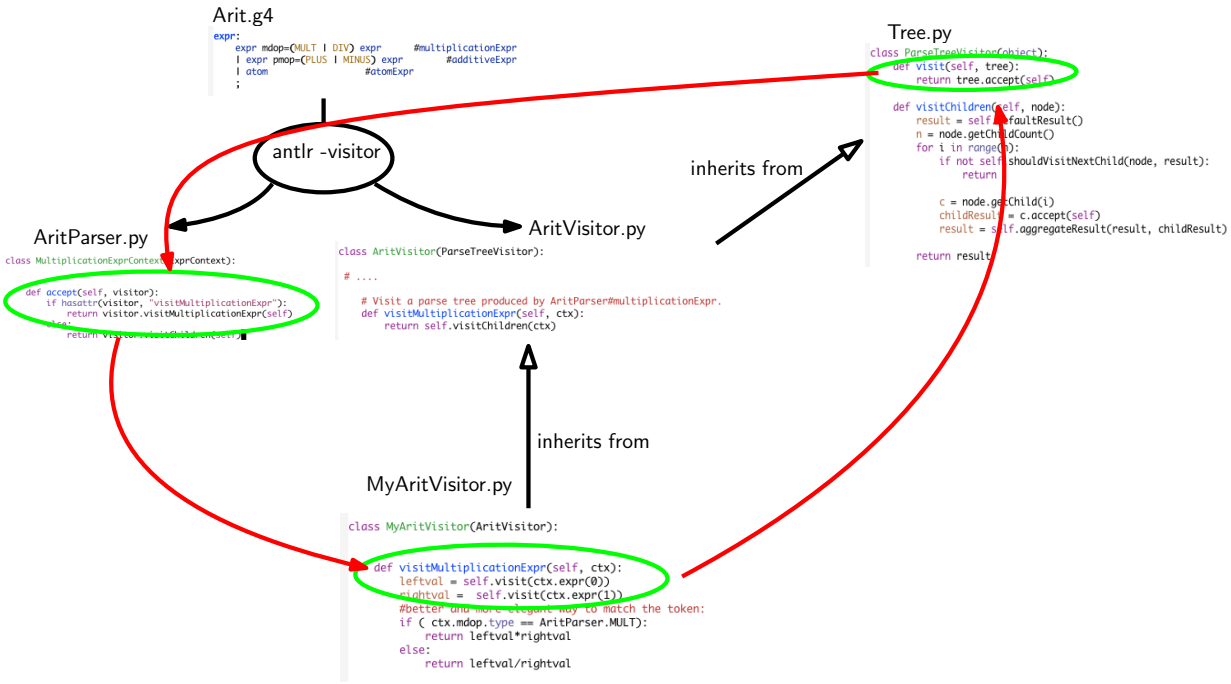

Figure 3.1: Visitor implementation Python/ANTLR4. ANTLR4 generates `AritParser` as well as `AritVisitor`. This `AritVisitor` inherits from the `ParseTreeVisitor` class (defined in `Tree.py` of the ANTLR4-Python library, use `find` to search for it). When visiting a grammar object, a call to `visit` calls 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.

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:

```
if ctx.addop.type == AritParser.PLUS:
    ...
```

To get the list of the two `expr` operands of the rule, you can use `ctx.expr()`. To get e.g. the second `expr` operand, you can use `ctx.expr(1)`. Be careful: if there is only one `expr` (for instance in the `visitParens` case) then `ctx.expr()` (and not `ctx.expr(0)` !) gives you the operand.

In other words, **what you can write in Python code is dictated by the `.g4` file**.

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 #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 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 (`MiniCC.py`) which will be the driver of our compiler:
  ```
  python3 MiniCC.py
  usage: MiniCC.py [-h] --mode {parse,typecheck,eval} [--debug]
                   [--disable-typecheck]
                   filename
  ```
  In this lab, you will use `-mode typecheck` or `-mode eval` The driver will launch the parsing of the input file, then the Typing visitor, and if the file is well typed, 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 code of the typer is completely given to you. The objective is here to read and understand code. The explicit questions are only to help you in this purpose. They will not be graded.

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 `TP03/MiniCTypingVisitor.py`.

Answer the following questions:

- Find the visitor base fonctions. How are their names constructed from the grammar rules (look at the g4 file) ?
- What are the basic types used in our Typer ?
- How is a typing exception handled ?
- Have a specific look at the assignment visitor ("ID=expr"): what is its name in the code ? What it is supposed to do ? Observe how it handles the type of the expression and the type of the variable.

Now predict the behavior of the typer on the following MiniC input:

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

type `make` to generate lexer, parser and visitor files. Then, launch the "compiler" in typer mode with:

```
python3 MiniCC.py --mode=typecheck TP03/tests/provided/examples-types/bad_type00.c
```

Observe the behavior of the visitor on all test files in `example-types/` **one by one**, and answer the following questions:

- What is the name of the function that checks for type equality ?
- 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? (*symbol table*)?

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

On incorrectly 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[3]:

```
make TEST_FILES='TP03/**/*type*.c' test
```

[3] If it takes time, you can *temporarily* remove the pyright rule in the Makefile.

As our makefile rule tests the whole typing and evaluator, it would fail on somes cases. We only test here "bad typing files", for which we already know their typing failure. You should obtain:

```
test_interpreter.py::TestInterpret::test_eval[TP03/tests/provided/examples-types/bad_type01.c] PASSED [ 16%]
test_interpreter.py::TestInterpret::test_eval[TP03/tests/provided/examples-types/bad_type_bool_bool.c] PASSED [ 33%]
test_interpreter.py::TestInterpret::test_eval[TP03/tests/provided/examples-types/bad_type04.c] PASSED [ 50%]
test_interpreter.py::TestInterpret::test_eval[TP03/tests/provided/examples-types/bad_type00.c] PASSED [ 66%]
test_interpreter.py::TestInterpret::test_eval[TP03/tests/provided/examples-types/bad_type03.c] PASSED [ 83%]
test_interpreter.py::TestInterpret::test_eval[TP03/tests/provided/examples-types/bad_type02.c] PASSED [100%]
```

If you get an error about the `--cov` argument, you didn't properly install `pytest-cov`.

Note on the -unit- tests The test infrastructure is implemented for you: we test your output and compare to the expected output. When another compiler is available (gcc, or a teacher version), we compare the different outputs.

The test infrastructure gives you a coverage score in order to evaluate how tests cover your code. Have a look on this score during the lab sequence.

The test infrastructure also uses pyright [4] to fast check typing annotations enabled in Python3 languages (cf `https://docs.python.org/3/library/typing.html`). These typing annotations are used in provided code to help you calling functions (enforcing the right types in function calls and returns).

Your own tests 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).

Note on printing library To allow compiling your MiniC programs with a regular C compiler (for tests, for instance), a `printlib.h` file is provided, and should be `#included` in all your MiniC test cases[5].

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.

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:

[4] `https://github.com/microsoft/pyright`)
[5] Note that unlike real C, `#` is a comment in MiniC to avoid actually having to deal with `#include`

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 3.3: 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 3.4: Interpretation for Statements

```
make
python3 MiniCC.py --mode=eval TP03/tests/provided/examples/test_print_int.c
```

and the interpreter will be run on `test_print_int.c` (it should print 42 on this particular exemple). **On the particular example `test_print_int.c` observe how integer values are printed.**

Open the interpretor source code(`MiniCInterpretVisitor.py`), and answer the following questions to understand the code:

- How is the memory initialised ? (compare to the memorytypes in the typer)
- Which values types can have a mini C expression (line 7)?
- How are string variables initialized in our interpreter?

TODO You still have to implement (in `MiniCInterpretVisitor.py`) [6]:

1. The modulo version of Multiplicative expressions (be careful to raise an exception if the operation is not valid).
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 test files `bad_xxx.c` for the expected error messages.
3. Statements: assignments, conditional blocks, tests, loops.

Test your implementation after each new feature:
```
python3 MiniCC.py --mode=eval /path/to/example/.c
```

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.

The file `MiniCInterpreter.py` in the skeleton already does this for you if you raise the right exception. These are also the values you will have to use in the `// EXITCODE` directives in your tests.

EXERCISE #8 ▶ **Automated tests for final delivery**
The rule `make test-interpret` launches everything. We recall that you can use the `TEST_FILES` makefile variable to launch only on a subset of your files.

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 the 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.

Test Infrastructure - a bit more explanation 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 them as special). The `EXITCODE` corresponds to the exit codes described in Section 3.3.2.

For instance, if you fail `test_print_int.c` because you printed `43` instead of `42`, using the command
```
make test TEST_FILES='TP03/tests/provided/examples/test_print_int.c'
```
you will get this error:

[6] Search for `NotImplementedError()`, they should all be removed for the source code.

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

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