Lab 3

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 `https://etudes.ens-lyon.fr` (NO EMAIL PLEASE)**, before **2024-10-10 23:59**. More instructions in section 3.6.

EXERCISE #1 ► **Lab preparation**
In the `cap-lab24` directory: `git pull` will provide you all the necessary files for this lab in `TP03` and `MiniC`. The latter folder will also be used for the next labs. ANTLR4 and `pytest` should be installed and working like in Lab 2, if not [1]:

```
python3 -m pip install pytest pytest-cov pytest-xdist
python3 -m pip install --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 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 #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 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 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`

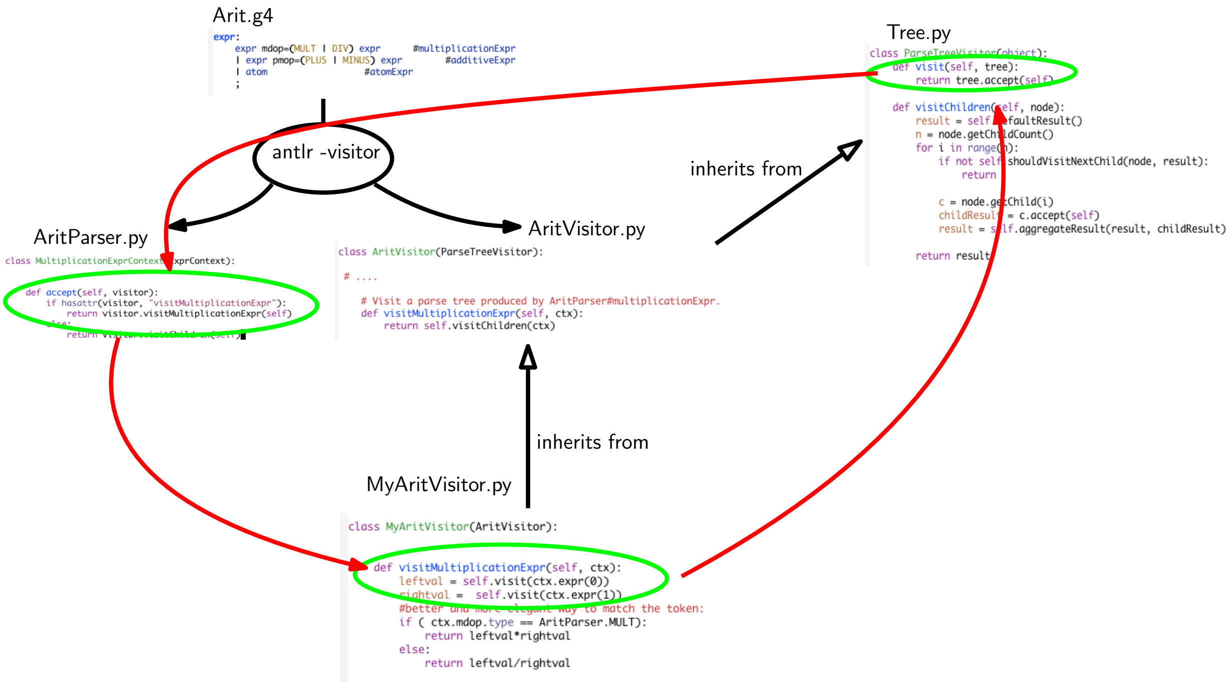

Figure 3.1: Visitor implementation Python/ANTLR4. ANTLR4 generates `AritParser` as well as `AritVisitor`. This `AritVisitor` inherits from the ParseTree visitor class (defined in `Tree.py` of the ANTLR4-Python library, use `find` to search for it). When visiting a grammar object, a call to `visit` calls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

3.1.2 Basic rules to write an ANTLR4 visitor

- 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)`.
- **what you can write in Python code is dictated by the `.g4` file.**

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 lab 2.

3.2 Up to you: first visitors

EXERCISE #3 ► **Trees (should be quickly done!)**
Consider the following grammar:

```
grammar Tree;


int_tree_top : int_tree EOF #top
    ;

int_tree: INT #leaf
    | '(' INT int_tree+ ')' #node
    ;


INT: [0-9]+;
WS : (' '|'\t'|'\n')+ -> skip;
```

This grammar represents "scheme-like trees", for instance `node (42 12 1515 17)` is the tree with root 42 and three children 12, 1515, 17.

1. We give you the grammar in the folder `tree/`. Copy and adapt previous files to get it operational.

2. Implement a visitor that decides whether a syntactically correct file is a binary tree. Your main file should contain:

```
tree = parser.int_tree_top()
visitor = MyTreeVisitor()
is_binary_tree: bool = visitor.visit(tree)
print("Is␣it␣a␣binary␣tree␣?␣" + str(is_binary_tree))
```

3.2.1 Application to MiniC Language

The objective is now to use visitors, to type and interpret MiniC programs, whose syntax is depicted in Figure 3.2. Classically, we should do typing first and the interpretation afterwards, but in this lab we will implement the interpretation first (assuming the program is well-typed).

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

EXERCISE #4 ► **Be prepared!**
In the directory `MiniC/` (outside `TP03/`), 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.
  - `python3 MiniCC.py --mode eval --disable-typecheck <file>` parses and interprets the given program, but does not typecheck it. This will be useful before you complete the typechecker implementation.

  Try it on some provided examples (e.g. in `TP03/tests/provided/examples-types/`), see what happens for well-typed and ill-typed programs (usually named `bad_*.c`).
- Two visitors to be completed: `TP03/MiniCTypingVisitor.py` and `TP03/MiniCInterpretVisitor.py`.
- Some test cases (`TP03/tests`), and a test infrastructure.

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).
- Conventions for division and modulo are the same as in C: division is truncated toward zero, and modulo is such that $(a/b)*b+a\%b=a$.

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

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

EXERCISE #6 ► **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:

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

Figure 3.4: Interpretation for Statements (pseudo-code)

```
make
python3 MiniCC.py --mode eval --disable-typecheck TP03/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).
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 lab3, but you will need it for 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 #7 ► **Automated tests**

We provide a script to automatically test your code. As at this point, you do not have your typechecker, modify the variable `DISABLE_TYPECHECK` to `True` in `test_interpreter.py`. You will have to put it back to `False` once you begin working on `MiniCTypingVisitor.py`.

Test with `make test` **and write an 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: 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. (Note that error messages raised from the typechecker have stricter formatting requirements, see below.)

Test Infrastructure Tests work mostly as in the previous lab. By default, the testsuite is ran on all `.c` files in the `TP03/tests/` directory. You may restrict to a set of files using `make test FILTER=....` `FILTER` is either a single file or an extended wildcard like `TP03/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 3.3.2.

For instance, if you fail `test_print_int.c` because you printed `43` instead of `42`, using the command

```
make test FILTER=TP03/tests/provided/examples/test_print_int.c
```

you will get this error:

```
____________TestInterpret.test_eval[TP03/tests/provided/examples/test_print_int.c] ____________

self = <test_interpreter.TestInterpret object at 0x7f05d6f86a40>,
filename = 'TP03/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
```

3.4 A type-checker for the MiniC language

3.4.1 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.

The expected errors of the typechecker are the following :

- "In function f: Line l col c: type mismatch for e: t1 and t2" for assignments and comparison (equality operands only), if the two arguments have different types;
- "In function f: Line l col c: invalid type for MESSAGE: t (and t')" for typing error, with MESSAGE explicit enough. For example: "In function main: Line 8 col 6: invalid type for multiplicative operands: integer and string";
- "In function f: Line l col c: MESSAGE" for errors that are not purely typing, e.g. undeclared variable or double declared variables. For example: "In function main: Line 5 col 2: Variable x already declared".

The name f is the current function, for the moment it should be 'main' but we may add functions later. Some of the tests provided, mainly of the form `bad_...c`, give examples of expected errors.

As before, we explicitly ask you to write new test cases, and make your error messages as explicit as possible.

3.4.2 Implementation of the Typechecker

EXERCISE #8 ► **Typing**

Write typing rules for expressions (on paper). Then, implement a type checker for the MiniC language[3] (as a standalone visitor `MiniCTypingVisitor`). We provide you with a (basic) class for basic types and the environment initialization with the declared types. The methods `_raise`, `_raiseNonType` and `_assertSameType` allow you to add informative exception handlers. The provided test files must guide you when the implementation cannot be directly derived from the typing rules. Testing is the same as for the interpreter, except that you do not have to put `--disable-typecheck` on your individual test.

Do not forget to modify the variable `DISABLE_TYPECHECK` back to `False` in `test_interpreter.py`, otherwise the command `make test` will not test your `MiniCTypingVisitor.py`!

Also, do not forget to intensively use new test files.

3.5 Language extensions

In this section, the instructions are all the same: for each new extension, implement the syntax, give new semantic rules (on paper), give new interpretation rules (code), new typing rules, relevant test cases, adapt the test infrastructure, ...

The maximum grade (20/20) correspond to a clear and documented code without any flaw, implementing at least one of the following extensions, and with a test suite of quality.

EXERCISE #9 ► **Fortran-like for loops**

Implement typing and interpretation for loops that look like the following example (static loop bounds, optional constant stride):

`k=42; for i=k to k+1515 by 2 { .... }`

Informal typing and semantics:

- The loop counter must be declared explicitly as int type before the loop ;
- `for i = a to b` is an empty loop if `b` is strictly smaller than `a` (except with negative stride) ;
- Stride can be any integer value. When null, the loop is infinite.

[3] We do not ask for a decorated AST, only type checking.

- Assigning the loop index within the loop is allowed, and when this happens the value assigned does not impact the next loop iterations (like Python's `for i in range(...):` loop).
- Loop bounds are evaluated when entering the loop, and not re-evaluated afterwards.

EXERCISE #10 ▸ **C-like for loops**
Extend the language with C-like for loops, with initial assignment, loop condition and increment assignment all optional. Example are the followings:
```
for (i=1;i<4;i=i+1) { .... }
for (;j<4.0;j=j+1.5) { .... }
```

3.6 Final delivery

We recall that your work is **personal** and code copy (including tests) is **strictly forbidden**.

EXERCISE #11 ▸ **Archive**
The interpreter and the typer (working together) are due on the course's webpage

`https://etudes.ens-lyon.fr/`

Type `make tar` in the `MiniC` folder to obtain the archive `MYNAME.tgz` to send (change your name in the Makefile before!). You have a (minimal) `README-interpreter.md` to fill with your name, the functionality of the code, how to use it, your design choices if any, the chosen extensions, and known bugs. Your archive must also contain your tests (TESTS!) in the `TP03/tests/students` folder. We expect unit tests (small files that test just one feature, or the interaction between a few ones, not everything at once in a huge test file) with clear and explicit names (typically `test_str_assign.c` and not `test04.c`). The command `make test` must work with your implementation; if some of the tests you have fail, please report the corresponding bugs in your readme. When there are some error messages given in examples and tests we provide, you are expected to write your code to produce exactly these messages.

You will be graded on 20 points, as follows:

- 10 points on the correctness of your code (do you pass all tests we could think of?),
- 5 points on the coverage of your tests (The coverage will be computed using your tests on both **your** and **our** implementations, the maximum grade will be obtained if your tests reach the same coverage as **our** reference test suite. Coverage is only considered for the main files of the lab (here: `MiniC.g4`, `MiniCInterpretVisitor.py` and `MiniCTypingVisitor.py`))
  ⟶ you can check your coverage by going to `MiniC/htmlcov/index.html`.
- 5 points on the correctness and coverage of the chosen extension (if any), the quality of your code and tests, your readme, your archive,...,
- if you deposit your work late, you will lose 1 point per hour of lateness,
- in case of plagiarism, if n students have the same code, then the grade of each student will be divided by n.