Lab 3
Interpreters and Types

Objective

- Understand visitors.
- Implement typers, interpreters as visitors.
- This is due **on `https://etudes.ens-lyon.fr` (NO EMAIL PLEASE)**, before **2021-10-04 23:59**. More instructions in section 3.6.

EXERCISE #1 ► **Lab preparation**
In the `cap-labs21` 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 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`

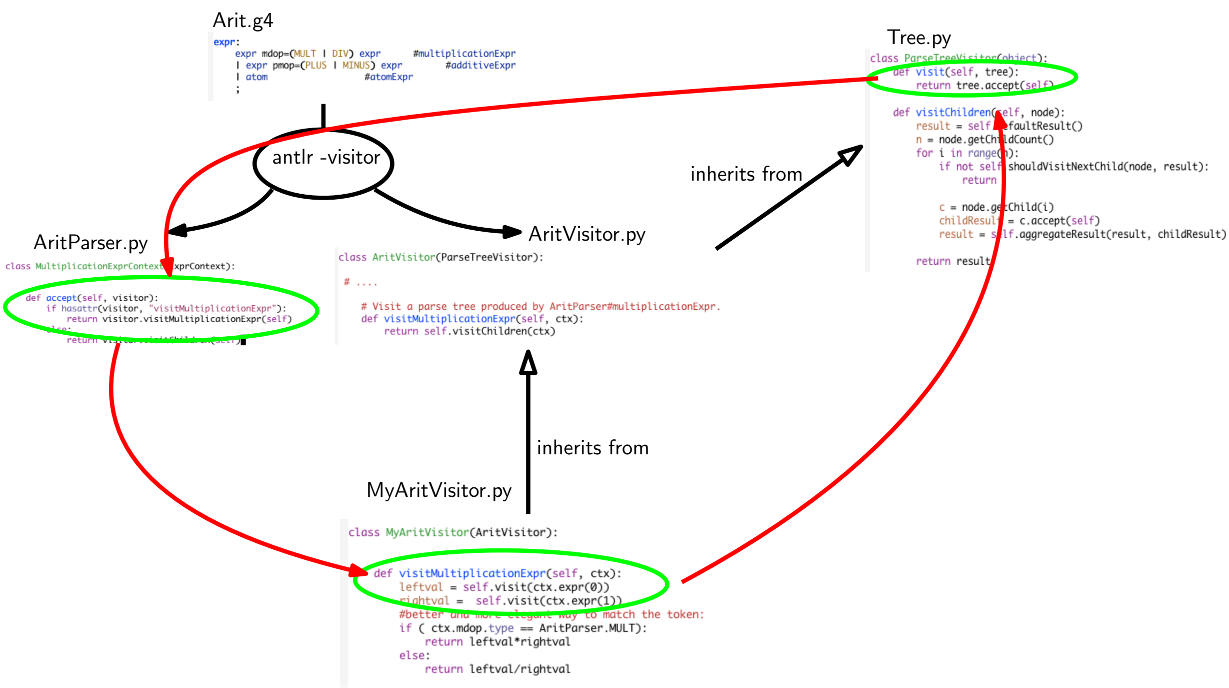

Figure 3.1: Visitor implementation Python/ANTLR4. ANTLR4 generates `AritParser` as well as `AritVisitor`. This `AritVisitor` inherits from the ParseTree visitor class (defined in `Tree.py` of the ANTLR4-Python library, use `find` to search for it). When visiting a grammar object, a call to `visit` calls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

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

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

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

```
if ( ctx.pmop.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 with in the `visitNotExpr` case) then `ctx.expr()` (and not `ctx.expr(0)` !) gives you the operand.

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()
b = visitor.visit(tree)
print("Is it a binary tree ? " + str(b))
```

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'.
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
    : PRINTLN_INT OPAR expr CPAR SCOL #printlnintStat
    | PRINTLN_FLOAT OPAR expr CPAR SCOL #printlnfloatStat
```

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`).
- 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.
- Two visitors to be completed: `TP03/MiniCTypingVisitor.py` and `TP03/MiniCInterpretVisitor.py`.
- Some test cases, 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).

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 #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.

Type:

```
make
python3 MiniCInterpreter.py TP03/tests/provided/examples/test00.c
```

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

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 test files `bad_xxx.c` for the expected error messages.
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 error.

EXERCISE #7 ▶ **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. (Note that error messages raised from the typechecker have stricter formatting requirements, see below.)

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, using the command
`make tests TEST_FILES='TP03/tests/provided/examples/test00.c'`
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 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:

```c
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) ;
- + 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".

f is the current function, for the moment it should be 'main' but we will add functions later.

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`) [4]. We provide you with a (basic) class for basic types and the environment initialization with the declared types. The methods `_raise`, `_raisemismatch` and `_raiseNonType` 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 have to give the value `True` to `enable_typing` in the file `MiniCInterpreter.py`.

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 code without any flaw, and implementing at least one of the following extensions.

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.
[4] Do not forget to enable the call to this visitor in the main file.

- 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.

EXERCISE #11 ► **Arrays**
We want to extend our mini language with imperative arrays. The syntax is augmented with the three following constructions:

- `Alloc(e)` allocates a new array of size equal to the value of e, with undefined values. By default, we only have arrays of int [5].
- `Read(e1,e2)` reads the e_2^{th} value of array e_1.
- `Write(e1,e2,e3)` modifies the e_2^{th} value of array e_1 with the value of expression e_3.

3.6 Final delivery

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

EXERCISE #12 ► **Archive**
The interpreter and the typer (working together) are due on the course's webpage

`https://etudes.ens-lyon.fr/course/view.php?id=4814`

Type `make tar` to obtain the archive to send (change your name in the Makefile before!). Your archive must also contain tests (TESTS!) and a (minimal) `README-interpreter.md` with your name, the functionality of the code, how to use it, your design choices if any, the chosen extensions, and known bugs.

[5] As an option, you can implement `Alloc<basetype>(e)`