Lab 3
Interpreters and Types

Objective

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

EXERCISE #1 ▶ **Lab preparation**
In the `cap-labs19` 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. The testsuite also uses `pytest-cov`, to be installed with:
`pip3 install --user pytest-cov`

3.1 Demo: Implicit tree walking using Listeners and Visitors

3.1.1 Error recovery with listeners

By default, ANTLR4 can generate code implemeting a Listener over your AST. This listener will basically use ANTLR4's built-in ParseTreeWalker to implement a traversal of the whole AST.

EXERCISE #2 ▶ **Demo: Listener (`Hello/`)**
Observe and play with the `Hello` grammar and its PYTHON Listener:
```
$ make
$ make run
<appropriate chain>^D
```

3.1.2 Interpret (evaluate) arithmetic expressions with visitors

In the previous exercise, we have traversed our AST with a listener. The main limit of using a listener is that the traversal of the AST is directed by the walker object provided by ANTLR4. So if you want to apply transformations to parts of your AST only, using listener will get rather cumbersome.

To overcome this limitation, we can use the Visitor design pattern[1], which is yet another way to seperate algorithms from the data structure they apply to. Contrary to listeners, it is the visitor's programmer who decides, for each node in the AST, whether the traversal should continue with each of the node's children.

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

EXERCISE #3 ▶ **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).

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

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.

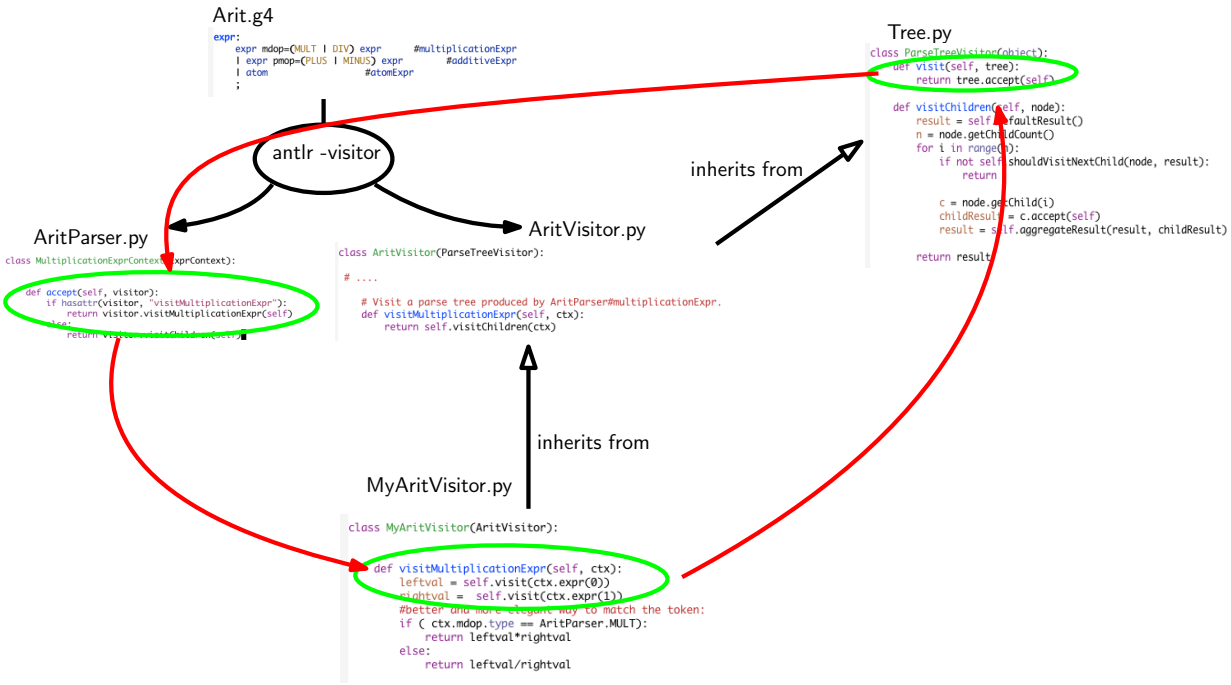

Figure 3.1: Visitor implementation Python/ANTLR4. ANTLR4 generates `AritParser` as well as `AritVisitor`. This `AritVisitor` inherits from the ParseTree visitor class (defined in `Tree.py` of the ANTLR4-Python library, use `find` to search for it). When visiting a grammar object, a call to `visit` calls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

3.2 Up to you: first visitors

EXERCISE #4 ▶ **Trees**

Consider the following grammar:

```
grammar Tree;


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. Write this grammar and test files.
2. Implement a visitor that decides whether a syntactically correct file is a binary tree. Your main file should contain:

```
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
    | if_stat
    | while_stat
    | print_stat
    ;

assignment: ID ASSIGN expr SCOL #assignStat;

if_stat: IF condition_block (ELSE IF condition_block)* (ELSE stat_block)? #ifStat;

condition_block: expr stat_block  #condBlock;

stat_block
    : OBRACE block CBRACE
    | stat
    ;

while_stat: WHILE OPAR expr CPAR  stat_block #whileStat;


print_stat
```

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

EXERCISE #5 ▶ **Be prepared!**
In the directory `MiniC-type-interpret/`, you will find:

- The MiniC grammar (`MiniC.g4`).
- A `Main.py` that parses the command line, does the lexical analysis and syntax analysis 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: `MiniCTypingVisitor.py` and `MiniCInterpretVisitor.py`.
- Some test cases, and a test infrastructure.

3.3 An interpreter for the MiniC-language

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.

c (literal)	`return int(c) or float(c)`
x (variable)	`find value 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. The typechecking will be implemented later. For now, you can reason in terms of "well-typed programs".

Type:
`make run TESTFILE='ex/testxx.c'`
and the interpreter will be run on `ex/testxx.c` (or on `ex/test00.c` if you do not specify variable `TESTFILE`).
On the particular example `ex/test00.c` observe how integer values, strings, boolean, floats values are printed.

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

1. Variable declarations (`varDecl`) and variable use (`idAtom`): your interpret 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 special value None for all variable declarations.** Refer to the three test files `ex/bad_defxx.c` for the expected error messages.

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

EXERCISE #8 ▶ **Unit 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. (Note that error messages raised from the typechecker have stricter formatting requirements, see below) To properly test the `ex/bad_def*` files, you will have to edit the python test script `test_interpreter.py`.

Test Infrastructure Tests work mostly as in the previous lab, with `// EXPECTED` and `// EXITCODE n` pragmas in the tests (be careful, it's now `//` for the comments, not `#`).

For instance, if you fail `test00.c` because you printed `42` instead of `99.00`, you will get this error:

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

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

self = <test_interpreter.TestCodeGen object at 0x7f0e0aa369b0>
filename = 'ex/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 '99.00\n1\n' == '42\n1\n'
E             - 99.00
E             + 42
E               1

test_interpreter.py:59: AssertionError
```

And if you did not print anything at all when `99.00` was expected, the last lines would be this instead:

```
        if expect:
```

```
>           assert(expect == eval)
E           assert '99.00\n1\n' == '1\n'
E             - 99.00
E               1

test_interpreter.py:59: AssertionError
```

3.4 A type-checker for the MiniC language

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.

EXERCISE #9 ▶ **Typing**
Write typing rules for expressions (on paper). Then, implement a type checker for the MiniC language[2] (as a standalone visitor `MiniCTypingVisitor`) [3]. We provide you with a (basic) class for basic types and the environment initialization with the declared types. The method `_raise`, `_raisemismatch` and `_raiseNonType` allows you to add informative exception handlers. The provided test files must guide you when the implementation cannot be directly derived from the typing rules.

The expected errors 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.

The interpreter must exit with exit code 1 whenever an error (typing or runtime) is encountered.

We explicitly ask you to write new test cases, and make your error messages as explicit as possible

EXERCISE #10 ▶ **Well typed program cannot go wrong**

Try your front-end on the following program:

```
int x;
y = x + 2;
```

Explain the slight difference with the result of the course.

[2] We do not ask for a decorated AST, only type checking.
[3] Do not forget to enable the call to this visitor in the main file.

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, ...

3.5.1 Mandatory language extension

EXERCISE #11 ▶ **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` ;
- Stride can be any integer value. When null, the loop is infinite.
- 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).

3.5.2 Optional language expressions

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

EXERCISE #12 ▶ **C-like for loops**
Extend the language with C-like for loops.

EXERCISE #13 ▶ **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 [4].
- `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.

EXERCISE #14 ▶ **Archive**
The interpreter and the typer (working together) are due on TOMUSS on Sunday, Oct 6th 2019, 6pm.

`http://tomuss-fr.univ-lyon1.fr`

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 `README.md` with your name, the functionality of the code, how to use it, your design choices, and known bugs.

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