Lab 3
Evaluators and Types

Objective

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

This Lab will last 3 sessions. Last one is Wednesday, Oct, 10th.

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

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 Evaluating 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 evaluator (`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. Override these methods in order to make them print the nodes' content by editing the `MyAritVisitor.py` file (use `print` instructions).

[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.

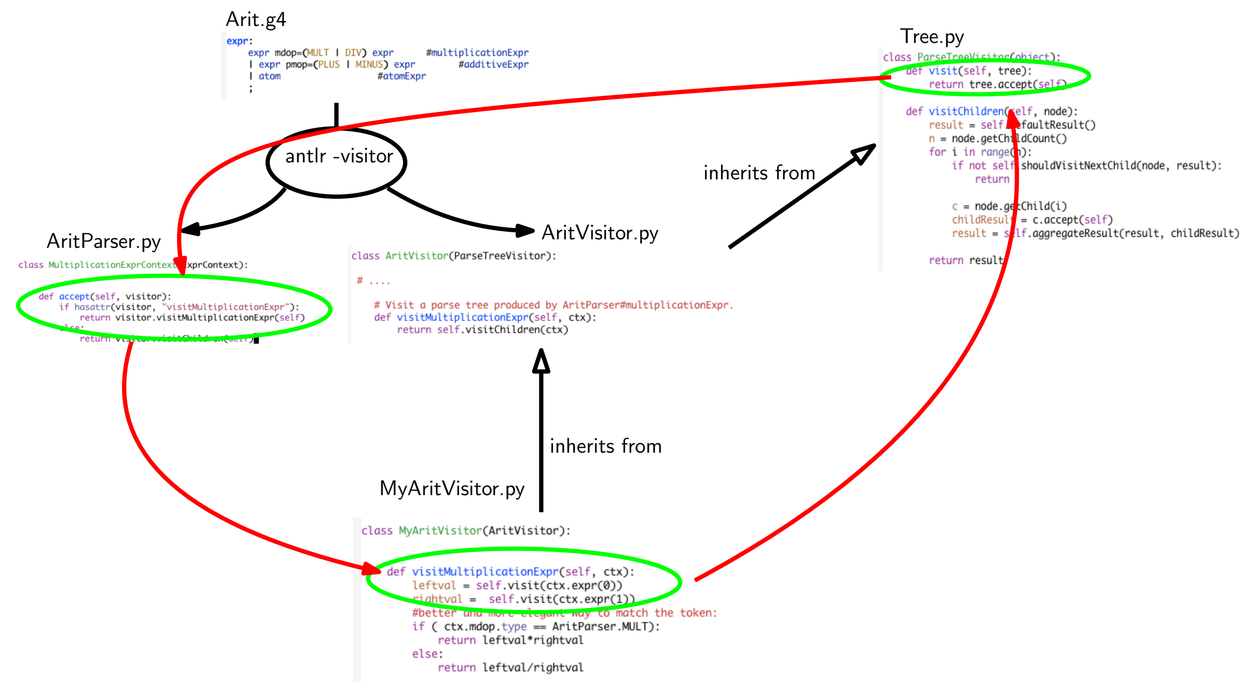

Figure 3.1: Visitor implementation Python/ANTLR4. ANTLR4 generates `AritParser` as well as `AritVisitor`. This `AritVisitor` inherits from the ParseTree visitor class (defined in `Tree.py` of the ANTLR4-Python library, use `find` to search for it). When visiting a grammar object, a call to `visit` calls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


tree:  INT     #leaf
    | '(' 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 evaluate Mu programs, whose syntax is depicted in Figure 3.2. A real compiler does the typechecking first and the evaluation afterwards, but in this lab we will implement the evaluation first (assuming the program is well-typed).

EXERCISE #5 ► **Be prepared!**
In the directory `Mu-evalntype/`, you will find:

- The Mu grammar (`Mu.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 Evaluator visitor.
- Two visitors to be completed: `MuTypingVisitor.py` and `MuEvalVisitor.py`.
- Some test cases, and a test infrastructure.

3.3 An evaluator for the Mu-language

The semantics of the Mu language (how to evaluate a given Mu 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 ► **Evaluator rules (on paper)**
First fill the empty cells in Figure 3.4, then ask your teaching assistant to correct them.

EXERCISE #7 ► **Evaluator!**
Now you have to implement the evaluator of the Mu-language. We give you the structure of the code and the implementation for numerical expressions and boolean expressions [2]. The typechecking will be implemented later. For now, you can reason in terms of “well-typed programs”.

Type:

```
make run TESTFILE='ex/testxx.mu'
```

and the evaluator will be run on `ex/testxx.mu` (or on `ex/test00.mu` if you do not specify variable `TESTFILE`). **On the particular example `ex/test00.mu` observe how integer values, strings, boolean, floats values are printed.**

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

1. Variable declarations (`varDecl`) and variable use (`idAtom`): your evaluator should use a table (*dict* in PYTHON) to store variable definitions and check if variables are correctly defined and initialized. Refer to the three test files `ex/bad_defxx.mu` for the expected error messages.

[2] We also implemented some “clever” behavious such as implicit casts in some expressions like string and int concatenation in +. You can have a look, or forget this (useless) feature.

```
grammar Mu;

prog: vardecl_l block EOF #progRule;

vardecl_l: vardecl* #varDeclList;

vardecl: VAR id_l COL typee SCOL #varDecl;

id_l
    : ID                #idListBase
    | ID COM id_l       #idList
    ;

block: stat*   #statList;

stat
    : assignment
    | if_stat
    | while_stat
    | log
    | OTHER {print("unknown␣char:␣{}".format($OTHER.text))}
    ;

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 expr stat_block #whileStat;


log: LOG expr SCOL #logStat;
```

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

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

EXERCISE #8 ► **Unit tests**

Test with `make tests` **and appropriate test-suite**. You must provide your own tests. The only outputs are the one from the `log` function or the following error messages: "`Undefined variable m`", "`m has no value yet!`". To properly test the `ex/bad_def*` files, you will have to edit the python test script `test_evaluator.py`.

Test Infrastructure Tests work mostly as in the previous lab. For instance, if you fail `test00.mu` because you printed 42 instead of `99.00`, you will get this error:

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

self = <test_evaluator.TestCodeGen object at 0x7f0e0aa369b0>
filename = 'ex/test00.mu'

    @pytest.mark.parametrize('filename', ALL_FILES)
```

e ::= c	`returns int(c) or float(c)`
e ::= x	`find value in dictionary and return it`

e ::= e_1+e_2	`let v1 = e1.visit() and v2 in e2.visit()`
	`if v1 and v2 are numbers (int, float)`
	`  return v1+V2`
	`else do some cast!`

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

Figure 3.3: Evaluation for expressions

```
    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_evaluator.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_evaluator.py:59: AssertionError
```

3.4 A type-checker for the Mu language

EXERCISE #9 ► **Typing**

Write typing rules for expressions (on paper). Then, implement a type checker for the Mu language[3] (as a standalone visitor `MuTypingVisitor`) [4]. We provide you with a (basic) class for basic types and the environment initialization with the declared types. The method `_raise` 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.

[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.

x := e	`let v = e.visit() in` `store(x,v) #update the value in dict`
log(e)	`let v = e.visit() in` `print(e) #python print`
S1; S2	`s1.visit()` `s2.visit()`
`if` *b* `then` *S*1 `else` *S*2	
`while` *b* `do` *S* `done`	

Figure 3.4: Evaluation for Statements

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

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 evaluation rules (code), new typing rules, relevant test cases, adapt the test infrastructure, ...

3.5.1 Mandatory language extension

EXERCISE #10 ▸ **Fortran-like for loops**
Implement typing and evaluation for loops that look like the following example (static loop bounds, optional constant stride):

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

3.5.2 Optional language expressions

EXERCISE #11 ► **C-like for loops**
Extend the language with C-like for loops.

EXERCISE #12 ► **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.

EXERCISE #13 ► **Archive**
The evaluator and the typer (working together) are due on TOMUSS on October, 16th, 2018, before 12pm (ie the night before next lab!):

`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.

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