Lab 3
An evaluator for the Mu language

Objective

- Understand visitors.
- Implement a simple evaluator as a visitor.

Todo in this lab:
- Play with visitors.
- Code a Mu Evaluator (section 3.3), which is due on tomuss on Nov 12, 2017.

EXERCISE #1 ▶ **Lab preparation**
In the `mif08-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 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 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
```

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

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

Also note the `#blabla` pragmas 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.

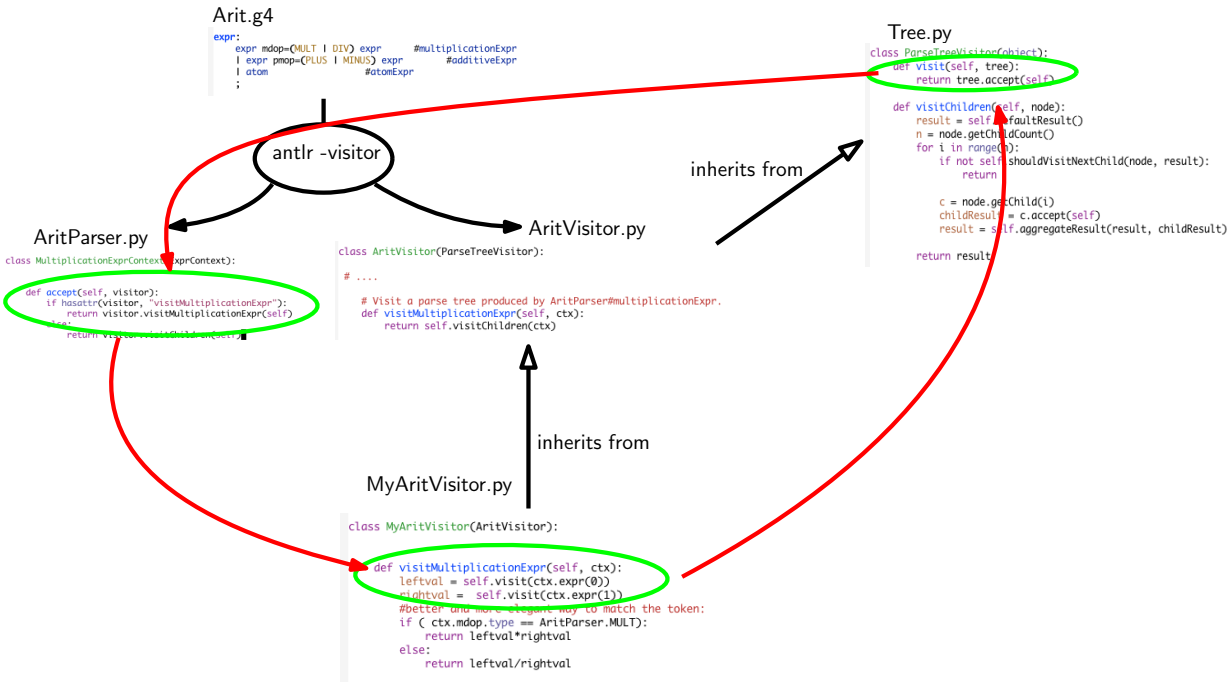

Figure 3.1: Visitor implementation Python/ANTLR4. ANTLR4 generates `AritParser` as well as `AritVisitor`. This `AritVisitor` inherits from the ParseTree visitor class (defined in `Tree.py` of the ANTLR4-Python library, use `find` to search for it). When visiting a grammar object, a call to `visit` calls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

We give you the Syntax of the Mu language, as a full grammar depicted in Figure 3.2. The objective is now to use visitors, firstly, to type Mu programs, and secondly, to evaluate them.

In the directory `Mu-evalntype/`, you will find:

- The Mu grammar (`Mu.g4`).

- A `Main.py` that parses the command line, do the lexical analysis and syntax analysis of the input file, then launch the Typing visitor, and if the file is well typed, launch the Evaluator visitor.
- Two visitors: `MyMuTypingVisitor.py`, which is complete, and `MyMuVisitor.py`, which is to be completed.
- Some test cases, and a test infrastructure.

3.2.1 A typing visitor for the Mu-language

EXERCISE #4 ► **Demo: play with the Typing visitor**
We provide you the code of the Typer for the Mu-language, whose objective is to implement the Typing rules of the course. Open and observe `MyMuTypingVisitor.py`, and predict its behavior on the following Mu file:

```
var x:int;
x="blablabla";
```

Then, test with:

```
make run FOO=ex-types/bad_type00.mu
```

Observe the behavior of the visitor on all test files in `ex-types/`. How do we handle:

- Multiplicative expressions with int and string operands ?
- Assignments to a variable which is not of the same type as the expression ?
- The variable type declarations ?

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

```
var x:int;
x="blablabla";

# EXPECTED
# Mismatch types for x
# EXITCODE 1
```

will be a successful unit test. Now, type:

```
make tests
```

and observe (Typing tests are those concerning files in `ex-types/`).

3.3 An evaluator for 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**
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]. As we provide you the typer, you can reason in terms of "well-typed programs".

Type:

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

[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.

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

You still have to implement (in `MyMuVisitor.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. 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)
    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
```

EXERCISE #9 ▶ **Archive**
The evaluator (all exercises in Section 3.3) is due on TOMUSS on November, 12, 2017, before 8pm. Type `make tar` to obtain the archive to send (change your name in the Makefile before!). Your archive must also contain tests and a `Readme.md` with your name, the functionality of the code, how to use it, your design choices, and known bugs.

```
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

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

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