Lab 3
An evaluator for the Mu language

Objective

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

First download the archive from the course website.

WARNING Exercice 3 (section 3.2) is evaluated (personal work only). Deadline Sunday, November 6th, 2016, 23h59, on TOMUSS.

3.1 Implicit tree walking using Listeners and Visitors

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 #1 ▶ **Listener**
Observe and play with the `Hello` grammar and its PYTHON Listener.

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 #2 ▶ **Arithmetic expression evaluator**
During the previous lab, you have implemented an expression evaluator with "right actions". Observe and play with the `Arit` grammar and its PYTHON Visitor. You can start by opening the `AritVisitor.py`, which is generated by ANTLR4: it provides an abstract visitor which you need to extend to build your own.

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.

We depict the relationship between visitors' classes in Figure 3.1.

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

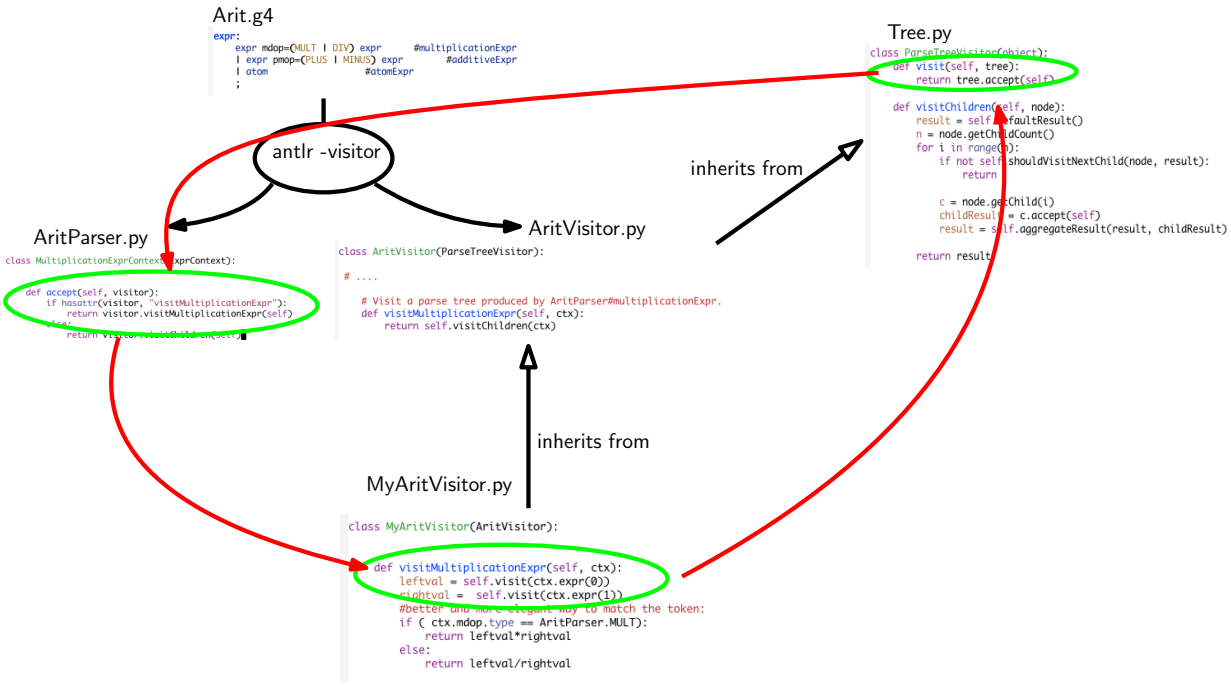

Figure 3.1: Visitor implementation Python/ANTLR4. ANTLR4 generates `AritParser` as well as `AritVisitor`. This `AritVisitor` inherits from the ParseTree visitor class (defined in `Tree.py` of the ANTLR4-Python library, use `find` to search for it). When visiting a grammar object, a call to `visit` calls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

3.2 An evaluator for Mu-language

We give you the syntax of the Mu language, as a full grammar depicted in Figure 3.2. 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 #3 ▶ **Evaluator!**

First fill the empty cells in Figure 3.4, then implement the evaluator for this mini-language. We give you the structure of the code and the implementation for numerical expressions and boolean expressions. For the moment, only consider well-typed expressions.

Your code, a makefile, a README and a bench of tests have to be uploaded on TOMUSS before the deadline. People with a "tiers-temps" have 24h supplementary delay. As usual, this is an individual work.

```
grammar Mu;

prog
 : block EOF
 ;

block
 : stat*   #statList
 ;

stat
 : assignment
 | if_stat
 | while_stat
 | log
 | OTHER {System.err.println("unknown␣char:␣" + $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`
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