Lab 3
Abstract Syntax Tree - Simple Evaluator

Objective

- Understand the notion of Abstract Syntax Tree (AST)
- Write a simple language evaluator with visitors.

Companion files are on the course git.

Mini-Project is due on Sunday, October 1st. 2017 8pm

3.1 Implicit tree walking using Listeners and Visitors

During the previous lab, you have implemented an expression evaluator using a parser with "right actions". In order to be able to do several traversal of the generated AST, but also in order to reuse the same parser for several application or target languages, one may actually be interested in decoupling the parsing phase from the evaluation. Listeners and Visitors are classes automatically generated by ANTLR4 that allows such features. They provide you with methods to navigates inside an AST and trigger actions according to the nodes encountered.

3.1.1 Error recovery with listeners

By default, ANTLR4 can generate code implementing 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 ► **Demo - Hello Listener**
Observe and play with the `Hello` grammar and its PYTHON Listener.

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 separate 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 ► **Demo: Arithmetic with visitors (`arith-visitor/`)**
Observe and play with the `Arit.g4` grammar and its PYTHON Visitor. You can start by opening 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.

Also note the `#blahblah` 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.

A last remark: when a ANTLR4 rule contains an operator alternative such as:

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

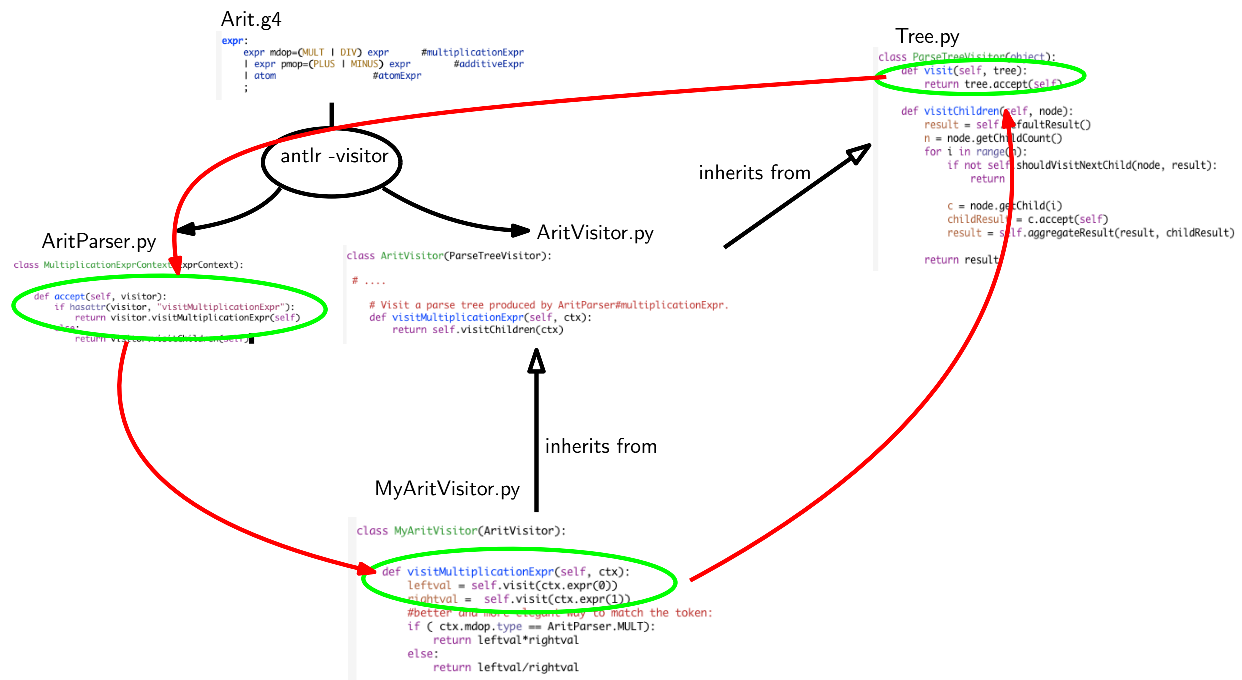


Figure 3.1: Visitor implementation in Python/ANTLR4. ANTLR4 generates `AritParser` as well as `AritVisitor`. This `AritVisitor` inherits from the ParseTree visitor class (defined in `Tree.py` of the ANTLR4-Python library, use `find` to search for it). When visiting a grammar object, a call to `visit`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

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

you can use the following code to match the operator:

```
      if ( ctx.pmop.type == AritParser.PLUS):
           ...
```

3.1.3 A tiny visitor

EXERCISE #3 ► **Trees**

Consider the following grammar:

```
grammar Tree;

tree: INT #feuille
    | NODE '(' INT tree+ ')' #arbre
    ;

INT: [0-9]+;
NODE: 'node';
```

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 syntactly 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))
```

3.2 Interpret a drawing mini-language

Credits This subject has been adapted from http://www.enseignement.polytechnique.fr/profs/informatique/Philippe.Chassignet/01-02/INF_431/dm_1/dm_1a.html

In this part, you will write an evaluator of a drawing language. The pictures to draw will be described in text files. Your program, `arit.py`, will take such a file, interpret it, print its evaluation in the terminal and draw the corresponding image in a Python Frame, unless the option –nopicts is set. Figures 3.2 and 3.3, show what is expected as output for each kind of statements and gives an example of drawing output.

```
12+ 2*3;
2 + 4 + 18 ;
(5+6) ;
print 100,100 for t = 0 .. 2;
set z = 1;
plot 100+t+2*z,100+2*t+42 for t = 0 .. 2;
quit;
```

(a) Definition

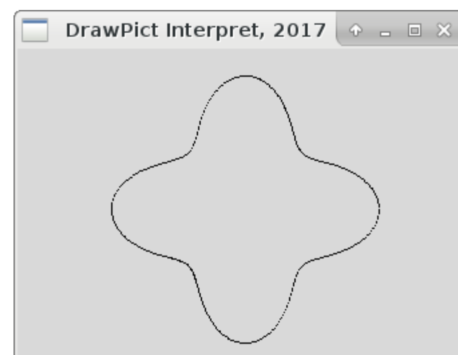
```
exprvalue = 18.00
exprvalue = 24.00
exprvalue = 11.00
(x,y)=(100.00,100.00)
(x,y)=(100.00,100.00)
(x,y)=(100.00,100.00)
z <− 1.00
plot (x,y)=(102.00,142.00)
plot (x,y)=(103.00,144.00)
plot (x,y)=(104.00,146.00)
```

(b) Output

Figure 3.2: A program with its corresponding stdoutput

```
plot 170+((4+cos(4*t))*cos(t)*20),120+((4+cos(4*t))*sin(t)*20) for
t=0 .. 2*pi by 0.01 ;
set x=2;
set z=x+pi+42;
print 100+t+2*z,100+2*t+42 for t = 0 .. 3;
quit;
```

(a) Definition



(b) Graphical print part: a Blop

Figure 3.3: A "blop" program with its interpretation, and an other program with its

3.2.1 The drawing language

A "picture file" is defined with respect to the ANTLR4 syntax defined in Figure 3.4.

Informal Semantics:

'AritPlot.g4'

```
grammar AritPlot;

prog:
     statement+ EOF #statementList
    ;

statement:
     expr SCOL #exprInstr
    | 'set' ID '=' expr SCOL #assignInstr
    | printplot=(PRINT|PLOT) expr ',' expr 'for' ID '=' expr '..' expr ('by' expr)?
       SCOL #printPlotInstr
    | 'quit' SCOL #quitInstr
    ;

expr:
    expr mdop=(MULT | DIV) expr #multiplicationExpr
    | expr pmop=(PLUS | MINUS) expr #additiveExpr
    | atom #atomExpr
    ;

atom:
     (INT | FLOAT) #numberAtom
    | ID #idAtom
    | '(' expr ')' #parens
    ;


PRINT : 'print';
```

Figure 3.4: Grammar for the mini-project

- a program is a sequence of instructions;
- an instruction is either an expression, a definition, a print, or a plot;
- an arithmetic expression is simply evaluated and print on the standard output. For instance `59+6` produces `exprvalue = 11.00`.
- a definition gives a way to save an expression into a variable (with the `set`) keyword;
- a 'print' prints a list of pairs on standard output: for instance,

  ```
  set z = 1;
  print 100+t+2*z,100+2*t+42 for t = 0 .. 3;
  quit;
  ```

  will produce the following output :

  ```
  z <- 1.00
  (x,y)=(102.00,142.00)
  (x,y)=(103.00,144.00)
  (x,y)=(104.00,146.00)
  (x,y)=(105.00,148.00)
  ```

  The (optional) keyword 'by' enables the free variable to be incremented of a (float) value (obtain from an arbitrary expression) different from one [2]. The last value (here, 3) is included in the for loop.
- a 'plot' draws on the graphical interface a polygonal chain going through the set of coordinates specified. It also prints the coordinates of each point to standard output. For instance,

  ```
  plot 100+t+2,100+2*t+42 for t = 0 .. 3;
  ```

[2] If the optional part of the `printPlotInstr` is non empty, then `ctx.expr(4)` is not None.

```
quit;
```
will execute the following drawing instructions:
```
printing line from (102.00,142.00) to (103.00,144.00)
printing line from (103.00,144.00) to (104.00,146.00)
printing line from (104.00,146.00) to (105.00,148.00)
```
and produce the following output:
```
z <- 1.00
plot (x,y)=(102.00,142.00)
plot (x,y)=(103.00,144.00)
plot (x,y)=(104.00,146.00)
plot (x,y)=(105.00,148.00)
```

3.2.2 Interpret!

EXERCISE #4 ► **Get code, play!**
In directory `arith-draw`:

```
make; make run
```

should print:

```
python3 arit.py ex/check01.txt
exprvalue = 18.00
```

by default we are in debug mode

EXERCISE #5 ► **Understand the test infrastructure**

```
make tests
```

compares, for each `.txt` file in the `ex` directory, the output of `arit.main` on stdout to the output describes in the comments part of the file.

For instance, if there is a test file named `exampletest.txt` containing:

```
12+ 2*3;
quit;
# EXPECTED
# exprvalue = 18.00
```

your `arit.main` will be called with this file name as argument, and is expected to print `exprvalue = 18.00` on stdout.

If your program prints something different, like `exprvalue = 18.001`, you will get an error of the form:

```
======================================================================
FAIL: test_ex (__main__.TestAritEval) [/path/to/your/repository/ex/exampletest.txt]
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_aritplot.py", line 30, in all_files_in_dir
    self.one_file(path)
  File "test_aritplot.py", line 76, in one_file
    self.assertEqual(actual, expect)
AssertionError: 'exprvalue = 18.001\n' != 'exprvalue = 18.00\n'
- exprvalue = 18.001
+ exprvalue = 18.00
?                  -


----------------------------------------------------------------------
```

EXERCISE #6 ► **Arithmetical expressions**

Extend the visitor for arithmetic expressions to be able to deal with the unary minus, the `pi` constant, floating point constants, cosinus and sinus operators, and extensively test with appropriate test files (expressions, and definitions). **Important remark: all numerical expressions are float or int but when printing, it is easier to print them as "float with two digits" and use `.2f` to print them.**

EXERCISE #7 ► **Print instruction**

On paper, define how to interpret a `print` instruction. Then implement the visitor function. Test. Retest.

EXERCISE #8 ► **Plot instruction**

On paper, define how to interpret a `plot` instruction in terms of "drawing lines" instructions. Then use the provided library to implement the visitor function. Test. Retest.

EXERCISE #9 ► **Homework**

Add a readme, make an archive (`make tar`), and email your work to your teaching assistants, with a subject containing `[CAP]`