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 website. **Project is due on Wed, October 12th.**

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 ▶
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.

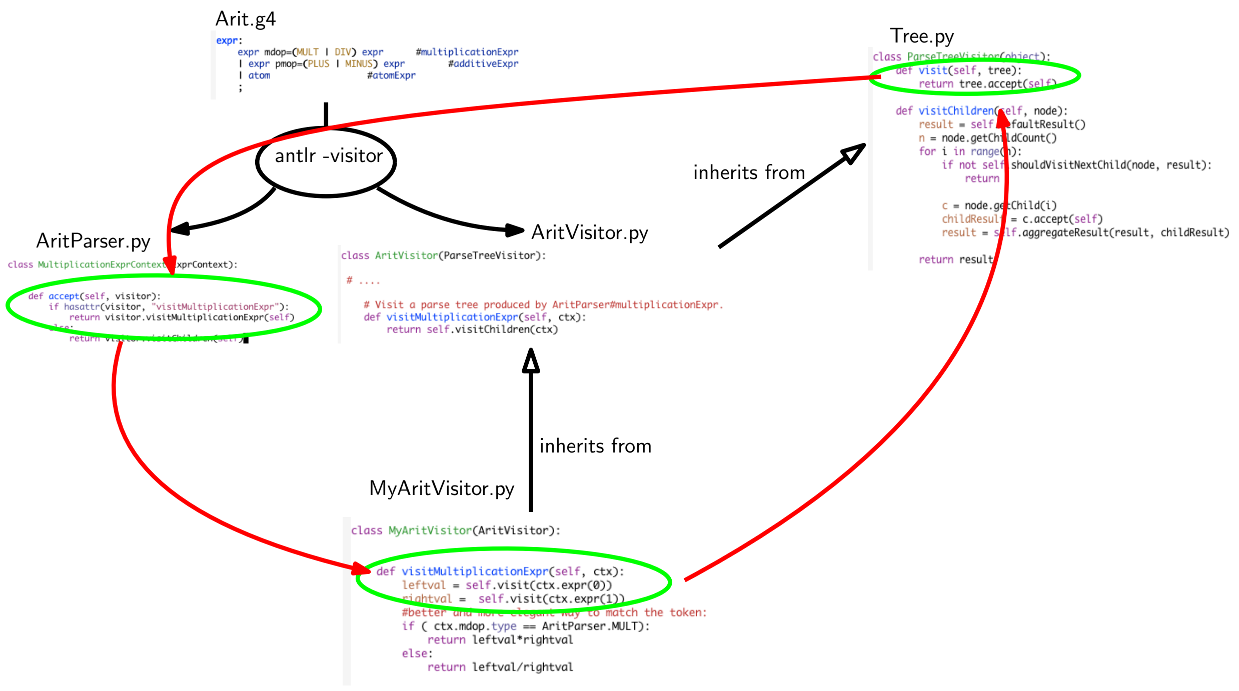
For every possible type of node in your AST, a visitor will implement a function that will apply to nodes of this type.

EXERCISE #2 ▶
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`



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

```
class ParseTreeVisitor(object):
    def visit(self, tree):
        return tree.accept(self)

    def visitChildren(self, node):
        result = self.defaultResult()
        n = node.getChildCount()
        for i in range(n):
            if not self.shouldVisitNextChild(node, result):
                return

            c = node.getChild(i)
            childResult = c.accept(self)
            result = self.aggregateResult(result, childResult)

        return result
```

```
class MultiplicationExprContext(ExprContext):

    def accept(self, visitor):
        if hasattr(visitor, "visitMultiplicationExpr"):
            return visitor.visitMultiplicationExpr(self)
        else:
            return visitor.visitChildren(self)
```

```
class AritVisitor(ParseTreeVisitor):

 # ....

   # Visit a parse tree produced by AritParser#multiplicationExpr.
   def visitMultiplicationExpr(self, ctx):
       return self.visitChildren(ctx)
```

```
class MyAritVisitor(AritVisitor):


   def visitMultiplicationExpr(self, ctx):
       leftval = self.visit(ctx.expr(0))
       rightval =  self.visit(ctx.expr(1))
       #better and more elegant way to match the token:
       if ( ctx.mdop.type == AritParser.MULT):
          return leftval*rightval
       else:
          return leftval/rightval
```

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.

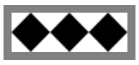
3.2 A patchwork evaluator - mini project

Credits This subject has been adapted from `http://www.enseignement.polytechnique.fr/profs/informatique/Philippe.Chassignet/02-03/INF_431/dm_1.html`.

In this part, you will write an evaluator of a Patchwork language [2]. The patchwork to draw will be described in a text file, and your program will interpret and draw in a Python Frame. For instance, the following program :

```
def losange = rot (b+rot rot rot b) + rot (rot b+rot rot b);
def ligne = losange[3];
def damier = (rot ligne)[3];
show damier;
```

will produce :



The patchwork language

A patchwork is defined with the following ANTLR4 syntax :

'PatchWork.g4'

```
grammar PatchWork;
```

[2] `http://en.wikipedia.org/wiki/Patchwork`

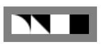
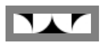
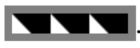
```
@header {
#header - global vars
}

@members {
# members
}

prog: instruction+ #ins
    ;

instruction:
        'show' draw ';'    #showDraw
    |   'size' draw ';'    #sizeDraw
    | 'def' ID '=' draw ';'#defDraw
    ;

draw:
        primitive         #primDraw
    | ID                  #idDraw
    | '(' draw ')'        #copyDraw
    | draw '[' INT ']'    #repeatDraw
    |  'rot' draw         #rotDraw
    | draw '+' draw       #concatDraw
    ;

primitive:
        'a'
    | 'b'
    | 'c'
    | 'd'
    ;

ID  :   ('a'..'z'|'A'..'Z'|'_')+ ;
INT :   '0'..'9'+ ;
WS  :   (' '|'\t'|'\n')+  -> skip;
```

Priorities : `[]` has precedence on `rot` which has precedence on + (concatenation).

Informal Semantics:

- a programme is a sequence of instructions;
- an instruction is either a definition, either a show, either a size
- a definition gives a way to save a patchwork into a variable, for instance:
  `def x = rot a;`
  `show x;`
  defines a new patchwork called `x` which value is `rot a`. Later, we use `x` as a patchwork.

a patchwork itself can be of the 5 following types:

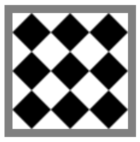
- the four primitives 'a', 'b', 'c' or 'd' (the little base squares) 'a' , 'b' , 'c' et 'd' ;
- an id (different from 'a', 'b', 'c' or 'd') that makes reference to a previous definition;
- a concatenation of two patchworks (+): a+b+c+d =  ;
- a rotation of a patchwork (rot is 90 degrées clockwise):a+rot a+rot rot a+rot rot rot a =  ; ;
- a repetition of a given patchwork: (b+d)[3] = .

EXERCISE #3 ▶ Interpret by hand!

An example is given in Figure 3.2. Represent on paper the different steps of the construction of the patchwork. Do all the intermediate steps.

```
def losange = rot (b+rot rot rot b)
               +rot (rot b+rot rot b);
def ligne = losange[3];
def damier = (rot ligne)[3];
show damier;
```

(a) Definition



(b) Interpretation

Figure 3.2: A program and its interpretation.

EXERCISE #4 ► **Pretty Printer Visitor**
Write a visitor `MyPatchworkPPVisitor.py` with the following model:

```
class MyPatchWorkPPVisitor(PatchWorkVisitor):
   def __init__(self):
      pass;
   def visitProg(self, ctx): #method to visit Prog rule
       n= len(ctx.instruction())
       print("Program␣with␣"+str(n)+"␣instructions.");
       for ins in ctx.instruction():
          self.visit(ins);
```

that pretty-prints a given patchwork file. Show that `rot a+b` is parsed as `(rot a)+b` and `a+b[7]` is parsed as `a+(b[7])`.

EXERCISE #5 ► **Evaluator as a visitor**
The file `LibPatchwork.py` contains all stuff to store and display the little squares. The file `TestLib.py` show you how to use it [3].

Now write a visitor that evaluates our language in terms of nice pictures. First of all, fill Tables 3.1 and 3.2. Then, write the appropriate Visitor.

Grammar element	Action
prog	create new Patchork evaluate all instructions
show `draw`	evaluate draw resize the Patchwork according to `draw`'s size call the Patchwork `showPicture()` method
size `draw`	
def `id` = `draw`	(use Python dictionaries)

Table 3.1: Actions at Instruction level

[3]launch with `python TestLib.py`

Grammar element	Action	Size
primitive ('a','b',...)	add (corresponding) square in patchwork at current (x, y) **be careful if the primitive is under a `rot`**	size=$(1,1)$
`ID` (in draw)		
`( draw )`	(should be easy!)	

`draw1+draw2` (concat)	**strongly depends on the rotation factor!**
	if this rotation is 0 :
	    evaluate `draw1` (at current (x,y,r))
	    evaluate `draw2` (at (x+length(`draw1`),y,r)
	else ...	

`rot draw`		
`draw[n]` (repeat)	evaluate `draw` once decide if repeat is horizontal or vertical (depends on rotation) compute the offsets accordingly (depends on `draw`'s size) evaluate (n-1) times with offsets set final length	size=(n*length(draw), height(draw))

Table 3.2: Actions at 'draw' level

For the code, we strongly encourage to :

- Add the following attributes to your visitor: x, y, r (position of the current picture, rotation), $sizex, sizey$ (sizes), $show$ (true if the 'draw' under evaluation will be drawn), and of course, pa (the current patchwork).
- First evaluate `draws` that do not contain variables;
- Compute only sizes and test intensively at this point;

- Compute drawing instructions for: primitives, repeat, rot, concat in this order;
- Def and use of variables in `draw` at the very end. (Use a table - dict in Python - to store the correspondance between id and draws)
- Intensively test, be careful to priorities.
- Write exceptions and handlers for user feedback.

For a little help, we give you our method implementation for "repeat" :

```
def visitRepeatDraw(self,ctx):
    n = int(ctx.INT().getText())
    print('repeat␣'+str(n)+'␣times␣')
    if (n<=0):  #todo exception here
        pass
    else:
        self.visit(ctx.draw())#do it once
        x,y=self._x,self._y
        l,h=self._sizex,self._sizey
        (decx,decy)=(0,0)
        if (self._r==0 or  self._r==2):
            decx=l #repeat horizontally
        else:
            decy=h #vertically
        #now repeat!!
        for i in range(1,n):
            self._x,self._y=(x+i*decx,y+i*decy)
            self.visit(ctx.draw())#do it but shifted
        #set final length
        self._sizex=n*l
        self._sizey=h
```

EXERCISE #6 ▸ **Homework**

Make an archive, a Readme, and send your work.