Writing Evaluators

MIF08

Laure Gonnord
Laure.Gonnord@univ-lyon1.fr

Lyon 1

Outline

1 Evaluators, what for?

2 Implementation

Analysis Phases

source code

↓

lexical analysis

↓

sequence of “lexems” (*tokens*)

↓

syntactic analysis (Parsing)

↓

abstract syntax tree (*AST*)

↓

semantic analysis

↓

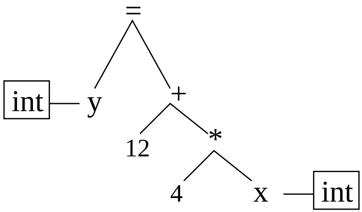
abstract syntax (+ symbol table)

Until now

We have parsed, and evaluate in semantic actions. But we want:

- more structure.
- an easier way to perform actions (not in the .g4 file).

Notion of Abstract Syntax Tree



- AST: memory representation of a program;
- Node: a language construct;
- Sub-nodes: parameters of the construct;
- Leaves: usually constants or variables.

Separation of concerns

- The semantics of the program could be defined in the semantic actions (of the grammar). Usually though:
  - Syntax analyzer only produces the AST;
  - The rest of the compiler directly **works with this AST**.
- Why ?
  - Manipulating a tree (AST) is easy (recursive style);
  - Separate language syntax from language semantics;
  - During later compiler phases, we can assume that the AST is **syntactically correct** $\Rightarrow$ simplifies the rest of the compilation.

Running example : Numerical expressions

This is an **abstract syntax** (no more parenthesis, ...):

$$
\begin{array}{rcll}
e & ::= & c & \textit{constant} \\
  & | & x & \textit{variable} \\
  & | & e + e & \textit{add} \\
  & | & e \times e & \textit{mult} \\
  & | & \ldots &
\end{array}
$$

Let us construct an AST to:

- Evaluate this expression (by tree traversal)
- Later: generate code for these expressions (by tree traversal)

Outline

1 Evaluators, what for?

2 Implementation
- Old-school way
- Evaluators with visitors

Outline

1. Evaluators, what for?
2. Implementation
   - Old-school way
   - Evaluators with visitors

Explicit construction of the AST

- Declare a type for the abstract syntax.
- Construct instances of these types during parsing (trees).
- Evaluate with tree traversal.

Example in Java 1/3

AST definition in Java: one class per language construct.

```
public class APlus extends AExpr {
    AExpr e1,e2;

    public APlus (AExpr e1,AExpr e2) { this.e1=e1; this.e2=e2; }

}
public class AMinus extends AExpr { ...
```

Example in Java 2/3

The parser builds an AST instance using AST classes defined previously.

ArithExprASTParser.g4

```
parser grammar ArithExprASTParser ;
options {tokenVocab=ArithExprASTLexer;}

prog returns [ AExpr e ] : expr EOF { $e=$expr.e; } ;

// We create an AExpr instead of computing a value
expr returns [ AExpr e ] :
  LPAR x=expr RPAR { $e=$x.e; }
| INT { $e=new AInt($INT.int); }
| e1=expr PLUS e2=expr { $e=new APlus($e1.e,$e2.e); }
| e1=expr MINUS e2=expr { $e=new AMinus($e1.e,$e2.e); }
;
```

Example in Java 3/3

Evaluation is an eval function per class:

AExpr.java

```
public abstract class AExpr {
    abstract int eval(); // need to provide semantics
}
```

APlus.java

```
public class APlus extends AExpr {
    AExpr e1,e2;
    public APlus (AExpr e1,AExpr e2) { this.e1=e1; this.e2=e2; }
    // semantics below
    int eval() { return (e1.eval()+e2.eval()); }
}
```

Outline

1. Evaluators, what for?
2. Implementation
   - Old-school way
   - Evaluators with visitors

Principle - OO programming

> *The visitor design pattern is a way of separating an algorithm from an object structure on which it operates.[...] In essence, the visitor allows one to add new virtual functions to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function.*

`https://en.wikipedia.org/wiki/Visitor_pattern`

Application

Designing evaluators / tree traversal in ANTLR-Python

- The ANTLR compiler generates a Visitor class.
- We override this class to traverse the parsed instance.

Example with ANTLR/Python 1/3

AritParser.g4

```
expr:
        expr mdop expr      #multiplicationExpr
    | expr pmop expr      #additiveExpr
    | atom                             #atomExpr
    ;

atom
    :   INT                  #int
    |   ID                   #id
    |   '(' expr ')'         #parens
```

- compilation with `-Dlanguage=Python2 -visitor`

Example with ANTLR/Python 2/3 -generated file

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


    # Visit a parse tree produced by AritParser#atomExpr.
    def visitAtomExpr(self, ctx):
        return self.visitChildren(ctx)

..
```

Example with ANTLR/Python 3/3

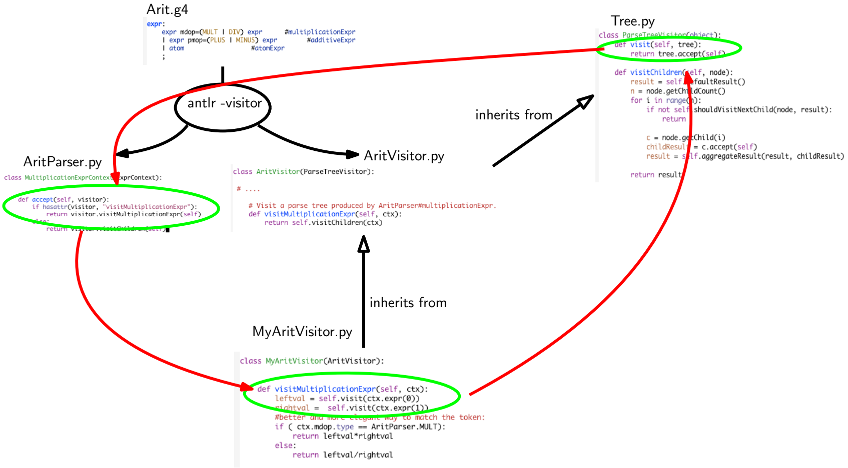
Visitor class overriding to write the evaluator:

MyAritVisitor.py

```
class MyAritVisitor(AritVisitor):
        # Visit a parse tree produced by AritParser#int.
    def visitInt(self, ctx):
        value = int(ctx.getText());
        return value;

    def visitMultiplicationExpr(self, ctx):
        leftval = self.visit(ctx.expr(0))
        rightval =  self.visit(ctx.expr(1))
        myop = self.visit(ctx.mdop())
        if ( myop == '*'):
            return leftval*rightval
        else:
            return leftval/rightval
```

Nice Picture (Lab#3)



From grammars to evaluators - summary

- The meaning of each operation/grammar rule is now given by the implementation of the associated function in the visitor.
- The visitor performs a tree traversal on the structure of the parse tree.