# EFREI Linguistique

# Project ANTLRv4

# Adrien Poupa

# Code selected: Scala

## 1. List of tokens

The following tokens are used:

```
MATH_OPERATOR: '+' | '-' | '/' | '*' | '<' | '>';
```
This is used to check if there is a mathematical operation involved.

```
TYPE: 'Double' | 'Complex';
```
The types used in the code snippet provided.

```
ARG_PREFIX_THAT: 'that';
ARG_PREFIX_VAL: 'val';
```
The prefixes that are used. I chose to separate them because some parts of the grammar needed only one of them.

```
DEF : 'def';
NEW : 'new';
OBJECT : 'object';
EXTENDS : 'extends';
OVERRIDE : 'override';
CLASS : 'class';
```
Classical keywords used in Scala language.

```
CONDITION : 'if' | 'else';
```
Check if we have a condition.

```
WS : [ \r\n\t] -> skip;
```
Skip spaces, tabs, newlines.

```
ID : [a-zA-Z]+;
```
Match lower-case identifiers.

```
NUMBER : [0-9]+;
```
Match number identifiers.

## 2. List of parser rules

```
program : object*;
```
Main rule to check the code against. There can be no program or has many as we need.

```
object: OBJECT ID EXTENDS ID '{' (class_def|val_def|call_function_def)*
'}';
```
Process the code: check that we have an object extending something that contains at least a class, a val or a function.

```
arg_function : ARG_PREFIX_THAT ': ' TYPE;
```
Arguments used by a function are like “that: type”.

```
function_def : DEF (ID|MATH_OPERATOR) '(' arg_function (',' arg_function)*
') ='
                ('{' statements_function '}'|statements_function);
```

The function itself: starts with “def”, has a name that is an ID or a math operator in the code provided, takes one or more arguments using the arg_function above followed by an equal symbol, and can or cannot have curly braces.

```
statements_function : (stmt_function)+;
```

We will have at least one statement within a function.

```
stmt_function :  NEW (ID|TYPE) '(' ((expr_function)+) ')'
        | ARG_PREFIX_VAL ID '=' ((expr_function)+)
        ;
```

A statement consists of either “new” declaration of an object of an identified type or an unknown identifier followed by parentheses that contain an expr_function or a variable declaration like “var myVar = …”. The “…” is a recursive call to expr_function.

```
expr_function :  expr_function((MATH_OPERATOR|'.') expr_function)
        | '(' ((expr_function)+) ')'
        | ','
        | ID
        | ARG_PREFIX_THAT
        ;
```

The core of the function: allows mathematical operations, parentheses, commas, identifiers or “that” keyword.

```
arg_class : ARG_PREFIX_VAL ID ': ' TYPE;
```

Arguments used by a class are like “val myVal: type”.

```
class_def : CLASS (ID|TYPE) '(' arg_class (',' arg_class)* ')' '{'
            (function_def|override_def)* '}';
```

A class starts with “class” followed by an identified type or an unknown identifier, the arguments between parentheses and the functions and/or overrides directives in the body of the class.

```
override_def : OVERRIDE DEF ID '=' (stmt_override)+;
```

An override instruction is like: “override def” followed by an unknown identifier and an equal sign, then a statement.

```
stmt_override : (expr_override)+;

expr_override :  '(' ((expr_override)+) ')'
        | '"' ((expr_override)+) '"'
        |  '.'
        |  '()'
        | CONDITION
        | ID
        | NUMBER
        | MATH_OPERATOR
        ;
```

An override can contain parentheses, quotes, a condition (if… else…), an unknown identifier, a number or a mathematical operator. It can also contain a dot or empty parentheses for function calls.

```
val_def : ARG_PREFIX_VAL ID '=' statements_val;
```

A val declaration is simple, like “val myVal = …”.

```
statements_val : (stmt_val)+;
stmt_val :  NEW (ID|TYPE) '(' ((expr_val)+) ')'
        ;

expr_val :  ','
          | ID
          | NUMBER
          ;
```

Here, we allow declarations like “new” followed by either a known type or an ID, then parentheses containing commas, IDs, empty parentheses for function calls or a number.

```
call_function_def : statements_call_function;

statements_call_function : (expr_call_function)+;

expr_call_function :  '(' (expr_call_function) ')'
        | expr_call_function (MATH_OPERATOR|'.') expr_call_function
        | ID
        | NUMBER
        | '()'
        ;
```

Finally, we can call a function from the class.

### 3. How the grammar works

The rule to code the test against is “program”.

It contains one or more “objects” which themselves can contain classes, values or calls to functions.

A class can contain functions or override directives.

Functions, classes, vals and calls to functions work the same way: a header rule, containing a statement rule calling an expr rule usually calling itself.

The whole grammar is working using this schema of Russian dolls: a large rule calling a smaller one, calling a smaller one, etc.

It is obviously not perfect, but I have tried to make it as restrictive as possible, hence the duplication of the “expr” rules. Having multiple rules allowed me to restrict them as much as possible.

### 4. Code examples

The snippets are provided in example1.scala and example2.scala.

### 5. Ideas

We could make the grammar more restrictive in order to make sure that a class called actually exists: for example, do not allow “new MySuperComplex” of MySuperComplex class does not exist.

We could also check if the prototype of the function is respected when it is called, so that “val y = new Complex(‘aa”, ‘bb’, ‘cc’)” would throw an error because of the wrong type (doubles vs strings) and number of parameter mismatch (2 vs 3).