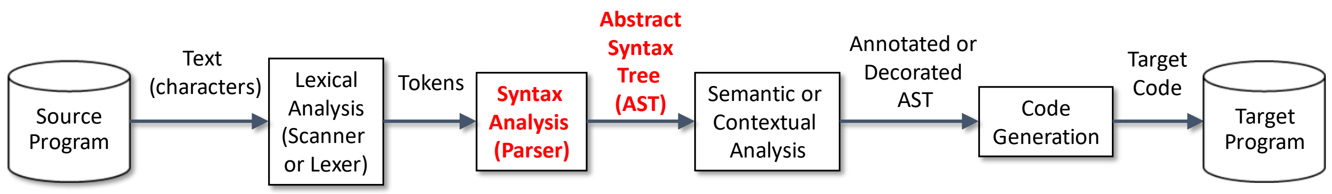
Programming Language Design
Department of Computer Science
Dr. Francisco Ortin
francisco.ortin@mtu.ie

# Laboratory 5 – Syntax Analysis

The objective of this laboratory is to implement the syntax analysis phase of the C-- language. It consists of the syntax recognizer (previous lab) and the creation of the corresponding AST, designed and implemented in lab 02.


Source Program
Text (characters)
Lexical Analysis (Scanner or Lexer)
Tokens
Syntax Analysis (Parser)
Abstract Syntax Tree (AST)
Semantic or Contextual Analysis
Annotated or Decorated AST
Code Generation
Target Code
Target Program

If no syntax error exists, the parser will recognize the input program and return its AST. Then, your language processor must display the AST created using Introspector.

## Remember

- ANTLR allows embedding Java code in grammar productions.
- Given an ID terminal symbol, the corresponding Token object in the embedded code is accessed via `$ID`.
- Given an `$ID` Token object, `$ID.getLine()` and `$ID.getCharPosition()+1` return, respectively, the line and column numbers of the `ID` token.
- Given an `expression` nonterminal symbol, its parse tree (`ExpressionContext`) can be accessed in the embedded code via `$expression`.
- Fields can be added to the nonterminal symbols in the grammar (e.g., `ast`), by adding a `returns` declaration in the production where the nonterminal symbol appears as the left-hand side of the production.
- When more than one instance of the same symbol occurs in the same production, use the `=` operator to rename the symbol and hence avoid ambiguity (e.g., `expression: e1=expression '+' e2=expression`).
- Notice: `$expression.start` returns the first token included in the derivation of the `expression` nonterminal symbol. Thus, `$expression.start.getLine()` returns its line number, although it can also be obtained with `$expression.ast.getLine()`.

## Go ahead

Create a new lab05 project and include the files in lab02 and lab04, along with the Introspector library (https://github.com/francisco-ortin/Introspector).

Add to Cmm.g4 the embedded Java code necessary to create the AST associated with each production. Each nonterminal symbol must have an `ast` field with its corresponding AST. In this way, the start symbol (program) will return the AST of the input program.

Using the input.txt file as a starting point, process its contents with your parser and load the resulting Abstract Syntax Tree (AST) into the Introspector. Verify that the AST's structure matches expectations. Create additional test files as needed to ensure your language processor is robust.

Introspector requires the implementation of the `toString` method in the AST nodes. Thus, implement `toString` in all the concrete classes of your AST. It should return a brief description of the syntactic construction it represents (an easy implementation). For program, just return the number of definitions; for function definition, its name and the number of parameters and statements.

Please, notice the following common mistake. Given the following definition "`int[8][3] fibonacci;`", the type of `fibonacci` *is array of 8 arrays of 3 integers* (instead of *array of 3 arrays of 8 integers*). Thus, the AST representing the type of `fibonacci` must reflect that correct type.