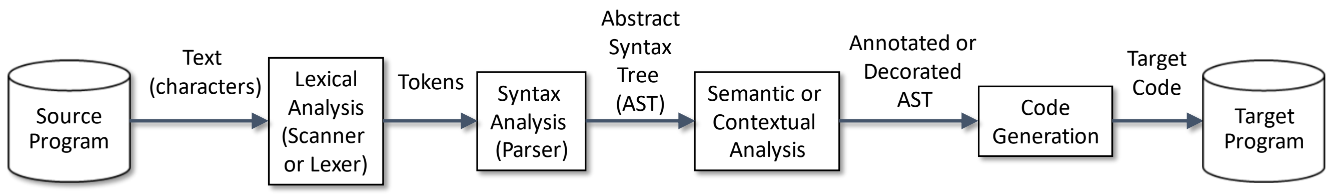
Programming Language Design
Department of Computer Science
Dr. Francisco Ortin
francisco.ortin@mtu.ie

Programming Language Design

In the laboratories of the Programming Language Design (PLD) module, we are going to design and implement a programming language called C--. The module assessment is project-based, so the compiler must be developed incrementally. This means that, if the student does not have the previous laboratory finished, the next one could not be done (in most cases). Two evaluations are undertaken: after finishing the syntax analyzer (lab 05), and at the end of the module (lab 12).

Laboratory 1 – Architecture of a Compiler

As we saw in lecture 1, compilers follow the Pipes and Filters architectural pattern. Each processing step is encapsulated in a filter component (compiler phase) that consumes data from its input and produces different data on its output. In our compiler, we have the following phases:


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

When developing a compiler, it is very important to know:

– The different phases of the compiler we are implementing.
– The responsibilities of each phase in the architecture.
– The input and output data of each phase.

All these elements were explained in the first lecture.

Mini-Compiler

A mini-compiler for a subset of the language to be implemented in this PLD module (C--) is given to the students.

– It provides the structure of the final compiler, just including a subset of the features of C--.
– The students may extend the mini-compiler to include the requested language features in forthcoming labs. In this way, this mini-compiler could be seen as a starting point for the student's work.
– It represents a practical way to understand the compiler architecture and the design patterns used in its implementation.
– In this laboratory, the student will just have to analyze the mini-compiler implementation and answer some questions about it. The answers will not be evaluated, but it is a very good exercise to understand language implementation.

Features of the Mini-Language

This mini-language has the following features:

- A mini-language program consists of a collection of global variable definitions, followed by a collection of statements in the main function (there are no local variables).
- The language only has two types: integer and real numbers.
- There are two statements: assignments and writing / printing in the standard output.
- Expressions could be created with variables, integer and real literals, parenthesis and the +, -, *, / and % arithmetic operators.

The following code is an example source program (provided as `input.txt`):

```
// Example program

int a;
double b;

void main() {
	a = 123;
	write 2*(a+2);
	b = 1.1;
	write b+2.2;
}
```

Compiling and Running Programs

We now describe how to compile and run programs in this mini-language. Although any Java IDE can be used, we encourage you to use IntelliJ Community, because it provides a wonderful ANTLR plugin (ANTLR is the lexer and parser generator tool we will use in this module).

First, create a new project / workspace. Second, copy all the files in the project. Third, include the antlr-4.*x*-complete.jar and introspector-*x*.jar as libraries (i.e., in the Java build- and class-path).

The compiler can be run the following ways (all the entry points are classes placed in the default package):

1. `Main` class. Runs the compiler with no debug information, receiving the input and output file names as parameters (e.g., `input.txt output.txt`).
2. `MainIntrospector` class. Like the previous one, but also shows the AST in a graphic window, using the Introspector tool. This is a useful tool to check the correct implementation of the language (the AST created for the input program).
3. `TestRigGUI` class. Shows the parse tree of the `input.txt` source program (useful when developing the parsers).
4. `TestRigTokens` class. Shows the tokens recognized in the `input.txt` source program (useful when developing the lexer).

5. `TestRigTrace` class. Displays how the syntax rules are derived in the parser, when the `input.txt` file is being parsed (a tool to debug the parser implementation).

If one of the first two options are executed, and the input file (e.g., `input.txt`) has no errors, then an output file (e.g., `output.txt`) is generated. The generated file is a low-level program in a stack-based intermediate language (called MAPL) similar to Java or .NET assembly languages. To run the `output.txt` file in MAPL, simply run the `run` script (`run.cmd` for Windows, and `run.sh` for Mac OS and Linux). Please, notice that .NET framework should be installed (Mono[1] works for Mac OS 10.14 and Linux).

The `antlr` script is used to generate the Java implementation of the lexer and parser (we use the ANTLR parser and lexer generator tool). Notice that this script does not compile a C-- program; for that purpose, you should run the compiler.

Go Ahead

Practice

Practice with the compiler:

– Compile the `input.txt` program and run the generated `output.txt` file using the MAPL virtual machine.
– Modify the program with other valid programs (see the language features above), and re-compile and re-run them.
– Include syntax errors to see how the compiler detects and shows them.
– Include semantic errors to see whether the compiler detects them or not.

Answer the following Questions

First, analyze the source code of the mini-compiler. Then, try to answer the following questions to check that you fully understand the language implementation (this is the main purpose of this first lab). Self-evaluation is provided with a document containing the answers, uploaded to Canvas (you do not need to send or upload your answers).

1. Identify the Java packages that implement the different phases of the compiler (lexical, syntax and semantic analysis, and code generation).
2. Which file describes the lexical specification of the language?
3. Which file describes the syntax specification of the language?
4. In which package(s) is (are) the Abstract Syntax Tree (AST) implemented?
5. What is the type (i.e., class or interface) that generalizes any node of the AST?
6. What is the concrete (non-abstract) AST node that represents the whole source program?

[1] If you use Mono, remember to pass the `--arch=32` command-line argument in Mac OS X. From macOS 10.15 Catalina on, 32 bits applications are no longer supported.

7. Do you recognize the design pattern used to model ASTs?
8. How many passes (AST traversals) are performed in the semantic-analysis phase?
9. What general type models/represents any AST traversal? Do you know the design pattern used?
10. Enumerate the different traversals defined in this particular implementation?
11. Which is the pass (traversal) that checks that variables must be defined before their use? Which phase does that traversal belong to?
12. Where is the main responsibility to infer the type of an addition expression placed? (i.e., what method should be modified if we want to change the current behavior)
13. Where is the responsibility of computing the MAPL size in bytes for each type in the source language placed?
14. In code generation, why are there two different classes for pushing values and addresses of expressions?
15. Which traversal directs the code generation process of the whole program (using other traversals)?