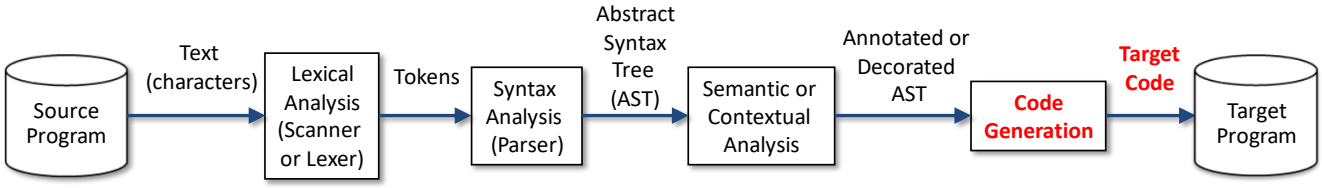
Programming Language Design
Department of Computer Science
Dr. Francisco Ortin
francisco.ortin@mtu.ie

Laboratory 11 – Code Generation I

In this lab, we will generate MAPL assembly code for a subset of C--. More precisely, we generate code for:

- Program, as a sequence of definitions.
- Read, write and assignment statements.
- Char, int and double literals (constants).
- Global variables.
- Arithmetic, comparison, logical and cast expressions.



Description

To generate code for the elements described above, we need to define the following code functions:

- Execute: translates high-level ASTs into a sequence of MAPL instructions with the same semantics as the high-level code. The execute code function is defined for statements, program and definitions.
- Value: translates an expression AST into the MAPL code to push its value onto the stack. This code function is defined only for expressions.
- Address: translates an expression AST into the MAPL code to push its memory address onto the stack. This code function is defined only for l-value expressions.

We must define the code templates for these three code functions. That process is similar to specifying the language semantics using denotations. In other words, denotational semantics can be used to implement the code generation phase of our compiler.

In lab 10, we implemented the `VisitorOffset` as part of the code generation phase. In this lab, we will add three new visitors, one for each code function: `ExecuteCGVisitor`, `ValueCGVisitor` and `AddressCGVisitor`.

Example

Let's see how to generate code for the Assignment AST node. Its abstract syntax is:

> Assignment: statement → expression$_1$ expression$_2$

Since assignments are statements, we must define the execute code generation template:

> execute[[Assignment: statement → expression$_1$ expression$_2$]] =
> address[[expression$_1$]]
> value[[expression$_2$]]
> <**store**> expresion$_1$.type.suffix()

It means that the code generated for an assignment is: the code for pushing the address of its left-hand side; followed by the code that pushes the value of its right-hand side; and then the appropriate `store` instruction to perform the assignment. The type of the `store` instruction (its suffix) is taken from the type of the expression on the left (the semantic analyzer annotated the AST with type information). Suffix is a new responsibility (method) to be added to types: it returns the type suffix used by many MAPL instructions.

The translation of the previous code template into the `ExecuteCGVisitor` is:

```
@Override
public Void visit(Assignment assignment, Void param) {
    assignment.getLeftHandSide().accept(this.addressCGVisitor, null);
    assignment.getRightHandSide().accept(this.valueCGVisitor, null);
    this.cg.store(assignment.getLeftHandSide().getType());
    return null;
}
```

It is advisable to place in another class (e.g., `CodeGenerator`) the low-level operations aimed at writing code in the output file. That class could be named `CodeGenerator` and `this.cg` in the code above is an instance of that class. Thus, `ExecuteCGVisitor` is focused on AST traversal, whereas `CodeGenerator` writes the low-level MAPL instructions. In this way, we decouple AST traversal from code generation.

We can also define the execute code generation template for program:

> execute[[Program: program → definition*]] =
> <**# source "input.txt"**>
> <**call main**>
> <**halt**>
> definition*.foreach(def -> execute[[def]])

To help you with the definition of code templates, an example input.txt C-- file and its corresponding ouput.txt MAPL code are provided.

Go ahead

- Define all the code templates for the C-- language elements identified at the beginning of this document. Remember to define the three code functions (execute, value and address).
- Implement the three code functions in three visitors: `ExecuteCGVisitor`, `ValueCGVisitor` and `AddressCGVisitor`.
- Make sure the generated programs are correct by running them with the MAPL VM, validating its runtime behavior.