# Compilers

## Project 5: Functions

*2nd Bachelor Computer Science*
*2022 - 2023*

Kasper Engelen
kasper.engelen@uantwerpen.be

For the project of the Compilers course you will develop, in groups of 2 students, a compiler capable of translating a program written in (a subset of) the C language into the LLVM intermediate language, and into MIPS instructions. You can also work alone, but this will make the assignment obviously more challenging. The project will be completed over the semester with weekly incremental assignments.

Concretely, you will construct a grammar and lexer specification, which is then turned into Python code using the ANTLR tool. The rest of the project will consist of developing custom code written in Python 3. This custom code will take the parse tree and turn it into an abstract syntax tree (AST). This AST will then be used to generate code in the LLVM IR language and in the MIPS assembly language. At various stages of the project you will also have to develop utilities to provide insight into the internals of the compiler, such as visualising the AST tree.

On Blackboard you can find two appendices: one in which an overview of the project is given, and one which contains some information on how to use ANTLR.

The goal of this assignment is to extend your compiler to support functions.

# 1 Functions

## 1.1 Grammar

Extend your grammar to support the following features, including the associated reserved keywords:

- (mandatory) Function scopes.

Your compiler should support function scopes. You also need to maintain support for the scopes you implemented in the previous assignment.

- (mandatory) Local and global variables.
- (mandatory) Functions.
  Your compiler needs to support functions. This includes:
  - Defining functions (including forward declarations).
  - Calling functions.
  - Passing parameters of basic types and pointer types. Note that such parameters may or may not be `const`. Pay attention to the difference between pass-by-value and pass-by-reference.
  - Returning values using the `return` keyword.
  - Functions that return nothing (i.e. the return type is `void`).

Example inputfile:

```
int mul(int x, int y){
        return x * y;
}

/*
 * My program
 */
int main(){
    int x = 1;
    while (x < 10) {
        int result = mul(x, 2);
        if ( x > 5) {
            result = mul(result, x);
        }
        printf(result); //show the result
        x = x + 1;
    }
    return 0;
}
```

### 1.2 Abstract Syntax Tree

You should construct an **explicit** AST from the Concrete Syntax Tree (CST) generated by ANTLR. Define your own datastructure in Python to construct the AST, such that the AST does not depend on the ANTLR classes.

### 1.3 Visualization

To show your AST structure, provide a listener or visitor for your AST that prints the tree in the Graphviz dot format. This way it can easily be visualized. For a reference on the dot format, see `http://www.graphviz.org/content/dot-language`. There are useful tools to open `dot` files such as `xdot` on Ubuntu, and Visual Studio Code on MacOS.

### 1.4 Semantic Analysis

Extend your symbol table and semantic analysis to support functions:

- (mandatory) Extend your symbol table to support function scopes.
- (mandatory) Check the consistency of the `return` statement with the return type of the enclosing function.
- (mandatory) Check the consistency between forward declarations and function definitions: return type, amount of parameters, type of parameters, etc.
- (optional) Check whether all paths in a function body end with a return statement (not required for procedures that return `void`).

### 1.5 Optimizations

Extend your optimization visitor to remove unreachable code and dead code:

- (mandatory) Do not generate code for statements that appear after a `return` in a function.
- (mandatory) Wherever you support the `break` or `continue` keywords, do not generate code for statements that appear after these keywords.
- (optional) Do not generate code for variables that are not used.
- (optional) Do not generate code for conditionals that are never true.

### 1.6 Code Generation: LLVM

Extend the code generation visitor for your AST that generates LLVM code and writes the generated code to a file. LLVM is an executable intermediary language used by popular compilers such as clang. Compiling to LLVM allows you to test your compiler early in the project, as it is closely related to the AST. More information on LLVM, as well as the language reference, can be found can be found on its website:

- `https://llvm.org`
- `https://llvm.org/docs/LangRef.html`