Compilers

Project 6: Arrays, Strings, and I/O

2nd Bachelor Computer Science
2023 – 2024

Kasper Engelen
kasper.engelen@uantwerpen.be

For the project of the Compilers course you will develop, in groups of 3 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.

1 Arrays and I/O

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

1.1 Functionality

Extend your grammar to support the following features:

- (mandatory) Arrays.
  Array variables should be supported, as well as operations on individual array elements. Mind the correct use of dimensions and indices.
  - (mandatory) multi-dimensional arrays.
  - (mandatory) assignment of complete arrays or array rows in case of multi-dimensional arrays.
  - (mandatory) array initialisation:

```
int values[3] = {3, 45, -9};
values = {-22, 7896, 1};
int other[3][4];
other[2] = {1,2,6,93};
```

  - (optional) dynamic arrays.

- (mandatory) Strings encoded as zero-terminated character arrays (i.e., C-strings). You should be able to parse string literals and assign them to character pointers (`char*`).

- (mandatory) Including `stdio.h`.
  Including `stdio` should be supported (`#include <stdio.h>`). This include will make the functions `int printf(char *format, ...)` and `int scanf(const char *format, ...)` available. If this include is not present, then using `printf` or `scanf` should result in an error.

  The `format` string allows interpretation of sequences of the form `%[width][code]` (width only in case of output). Provide support for at least the type codes `d`, `x`, `s`, `f` and `c`. Also provide support for `%%`. You may consider the `char*` types to be `char` arrays. Flags and modifiers do not need to be supported. Other includes or imported functions do not need to be supported either.

  Note: even though you already provided support for `printf` in a previous assignment, this will provide more functionality.

  Note: You do not have to support including the actual `stdio.h` header. If `stdio.h` is included, you can simply hard-code the `printf` and `scanf` code.

  Note: You may assume the `printf` statement to contain a string literal. You do not have to support formatting strings of the type `char*`.

Example input file:

```
#include <stdio.h>

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("%d", 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

You should implement the following checks:

- (mandatory) Type checking for array types (e.g., when passed as function parameters, when assigned).
- (mandatory) You should check when an array is accessed that the type of the specified index is an `int`.
- (mandatory) When assigning array initialisers (`{ ... }`), you should check that the length of the initialiser matches that of the array.

1.5 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`