Compilers

Project 7: Advanced Features

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.

The goal of this assignment is to extend your compiler to support some **advanced features**: structs, unions, file reading, and file writing.

1 Advanced features

1.1 Functionality

Extend your grammar to support the following features:

- (mandatory) User-defined structs. The structs should support members with primitive fields, arrays, enum types, as well as pointer types. You do not have to support default values for the struct members.

```
struct ListNode {
    int value;
    char node_name[10];
    struct ListNode* next_ptr;
};
```

- (optional) Structs that contain other structs as a value.

```
struct Packet_Header {
    int src_addr;
    int protocol;
    char flags[4];
    enum PacketType type;
    struct Packet data; // not a pointer
};
```

- (optional) Dynamic allocation of structs.
- (optional) Unions.

```
union NotTypeSafe {
    int as_integer;
    char as_str[30];
    float as_float;
};
```

- (optional) Function pointers. All the checks and errors that apply to ordinary pointers should also be implemented for function pointers.

```
int increment(int x, float f) {
    return  x + 1;
}

int main() {
    int (*incrementPtr)(int, float);
    incrementPtr = &increment;
    int z = 5;
    z = (*incrementPtr)(z, 0.689);
    // z is not equal to 6

    return 0;
}
```

- (optional) File reading using `fgets`. You can implement file pointers (`FILE*`) together with `fopen` and `fclose`, but this is not required. You may also just pass filenames to `fgets`:

```
char* read_file() {
    char buffer[50];
    fgets(buffer, 50, "some_filename.txt");

    return buffer;
}
```

- (optional) File writing using `fputs`. You can implement file pointers (`FILE*`) together with `fopen` and `fclose`, but this is not required. You may also just pass filenames to `fputs`:

```
void write_file() {
    fputs("new content for the file", "some_file.txt");
}
```

- (optional) Dynamically allocated strings and character buffers. These can work in combination with `printf`, `fgets`, `fputs`, etc.

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 function pointers: assigning function pointers, calling functions, etc.
- (mandatory) Type checking for accessing and assigning struct members.
- (mandatory) Type checking for accessing and assigning union members.

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`