Compilers

Project 4: Loops, Conditionals, and Scopes

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 Loops, Conditionals, and Scopes

The goal of this assignment is to extend your compiler to support loops and conditionals.

1.1 Grammar

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

- (mandatory) Conditional statements: `if` and `else` must be supported. You can assume that curly braces are required.
- (optional) `else if` statements.
- (mandatory) Loops. You have to implement `while`, `for`, `break`, and `continue`. **Hint:** You can implement for-loops by translating them to while-loops during the construction of the AST.
- (mandatory) Anonymous scopes.
  Your compiler should support unnamed scopes, next to the scopes from loops and conditionals (see the example below).
- (mandatory) You can also provide support for switch statements. For this you have to implement `switch`, `case`, `break`, and `default`. **Hint:** You can implement switch statements by translating them to if-statements in the AST.

  Note: the switch statement does not have its own scope! As such, you normally cannot declare variables inside a switch statement. If you do want to declare variables in a switch statement,

you first have to declare an anonymous scope. You can best implement this as a semantic error, and check for this after parsing.

- (mandatory) Your compiler should support enumerations.

  ```
  enum <enum_name> {
      <label 1>,
      <label 2>,
      ...
  };
  ```

  Note: In the C-language, an `enum` is actually an integer. In your compiler you should implement the `enum` values as `int` values.

  Note: You do not have to support assignments to constants. You may assume the first constant is zero, the second constant is one, etc. You do, however, have to support operations on the constants. They behave identically to integer variables.

Example input file:

```
/*
 * My program
 */

enum SYS_IO_ReceiverStatusBit {
    READY,
    BUSY,
    OFFLINE
};

int main() {
    enum SYS_IO_ReceiverStatusBit status = BUSY;
    {  // unnamed scope
        int x = 1 + status + OFFLINE; // Note: unscoped access of enum constants!
        while (x < 10) {
            int result = x * 2;
            if ( x > 5) {
                result = result * x;
            }
            printf("%d", result); //show the result
            x = x + 1;
        }
    }
}
```

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 to support scoping. Note that it is now possible to have the same identifier in different scopes. When resolving an identifier, first the current scope should be checked, before checking parent scopes.

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`