Compilers

Project 3: printf, comments, LLVM

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 Output and comments

The goal of this assignment is to add the `printf` function, comments, and LLVM IR code generation, so that your compiler can output to the LLVM intermediate representation.

1.1 Grammar

Extend your grammar to support the following features:

- (mandatory) In assignment you added a rule to your grammar to ignore comments. For this assignment you will have to make the following adjustments:
  - (mandatory) Support for single line comments and multi line comments.
  - (mandatory) Instead of simply ignoring comments, you can increase the readability of the generated LLVM code by retaining the comments from the input code during the compilation process. The comments will thus be stored in the AST. Such comments can then be put into the LLVM code.
  - (mandatory) You can increase the readability of the generated code even further by adding, after every instruction, a comment that contains the original statement from the input code. Note that, if a C statement translates to multiple LLVM instructions, the comment should only be added after the first LLVM instruction that corresponds to that C statement.
  - (mandatory) Implement the two points above also for the **MIPS output** that you will

have to implement later in the project.

- (mandatory) Outputting to the standard output using `printf`.
  In order to test the generated LLVM code, we want to be able to print out variable values. Since we do not yet support function calls, you can (for now) hardcode the `printf` function in your grammar as:

  ```
  'printf' '(' "%s" ',' (var | literal) ')'
  'printf' '(' "%d" ',' (var | literal) ')'
  'printf' '(' "%x" ',' (var | literal) ')'
  'printf' '(' "%f" ',' (var | literal) ')'
  'printf' '(' "%c" ',' (var | literal) ')'
  ```

  Note that this temporary implementation does not yet provide all the formatting options of `printf`, for now you may hardcode the formatting strings that are listed above in the grammar. The above formatting strings are for strings, integers, floats, and characters, respectively. Note that you also do not yet have to support strings for the moment.

- (mandatory) You should support typedefs. These allow you to create "new" types by assigning custom names to existing types. The newly created type must not have the name of an existing type!

  ```
  typedef <old name> <new name>
  ```

Example input file:

```
/*
 * My program
 */

typedef float speed;

int main() {
        int x = 5*(3/10 + 9/10);
        speed y = x*2/( 2+1 * 2/3 +x) +8 * (8/4);
        float result = x + y; //calculate the result
        printf("%f", result); //show the result
}
```

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.

2 Code Generation: LLVM

Implement a 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 the LLVM website:

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