Compilers

Project 1: Expressions

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 Expression Parser

The goal of this assignment is to implement a parser for mathematical expressions, and to construct and visualize an AST representation of these expressions.

1.1 Functionalities

Construct a grammar for simple mathematical expressions, operating only on `int` literals. Every expression should end with a semicolon. Input files can contain multiple expressions.

The following operators must be supported:

- (mandatory) Binary operations `+`, `-`, `*`, and `/`.
- (mandatory) Binary operations `>`, `<`, and `==`.
- (mandatory) Unary operators `+` and `-`.
- (mandatory) Parenthesis to overwrite the order of operations.
- (mandatory) Logical operators `&&`, `||`, and `!`.
- (mandatory) Comparison operators `>=`, `<=`, and `!=`.
- (mandatory) Binary operator `%`.

- (mandatory) Shift operators `<<`, `>>`.
- (mandatory) Bitwise operators `&`, `|`, `~`, and `^`.

Example inputfile:

```
5*(3/10 + 9/10);
6*2/( 2+1 * 2/3 +6) +8 * (8/4);
(1
+
1);
```

Notes:

- Make sure to differentiate between lexer rules and parser rules in your grammar. Lexer rules are defined in uppercase (e.g. "`LEXER_RULE`") while parser rules are defined in lowercase (e.g. "`parser_rule`"). The lexer rules will split the source file into tokens using regexes. The parser rules will then arrange these tokens into a parse tree.
- Make sure that your grammar is easily extendable. For example, while you currently only have to support integer literals, your final C compiler will also have to support other types as well. Make your grammar general enough such that adding new types of literals, operations, etc. can be done without drastic changes. This will save you a lot of time later on!
- You can ignore whitespace in your input files using the following rule in your grammar:

  ```
  WS: [ \n\t\r]+ -> skip;
  ```

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.

Notes:

- The goal of the AST is to maintain only the necessary information from the CST. For example, there is no need to store brackets in the AST as the structure of the tree already forces the order of operations.
- Inheritance can be used for the different types of nodes in the AST, which will make it easier to expand your compiler later on.
- You can implement the visitor pattern for your own tree datastructure to allow easy traversal of your AST.

1.3 Constant Folding

Constant expressions, such as the ones parsed in this assignment, can be evaluated at compile time. Hence, most compilers will not actually generate machine code (assembler) for these kinds of expressions. Rather, they will replace these expressions in the AST with a literal node containing the result.

Implement an optimization visitor that replaces every binary operation node that has two literal nodes as children, with a literal node containing the result of the operation. Similarly, it should also replace every unary operation node that has a literal node as its child, with a literal node containing the result of the operation.

It can be useful to implement a flag that allows you to turn off the constant folding, so that during testing you can more easily test the compiler.

1.4 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.

Notes:

- Visualising the AST is useful when building your compiler, as it can be used to debug the grammar and parser.