Compilers

Project 1: Expressions

2nd Bachelor Computer Science 2022-2023

Kasper Engelen
kasper.engelen@uantwerpen.be

For the project of the Compilers-course you will develop, in groups of 2 students, a compiler capable of translating a program written in a subset of C 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.

For the compiler you will construct a grammar and a lexer specification, which is then turned into a fully working parser using the ANTLR tool. You will then have to write code (in Python) to translate this parse tree into an abstract syntax tree (AST). Finally, you will have to write code that will translate such an AST into the LLVM and MIPS languages. Keep in mind that the programmer might make programming errors in C, and that your compiler needs to be able to deal with such errors.

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 Installation and usage of ANTLR and Python bindings

The ANTLR tool (`http://www.antlr.org`) can be used out-of-the-box as a Java jar file. The latest version can be downloaded from `https://www.antlr.org/download`. It requires Java to run.

ANTLR converts a grammar to Python classes using the following command:

```
java -jar antlr-4.12.0-complete.jar -Dlanguage=Python3 MyGrammar.g4
-visitor
```

In the example, `MyGrammar.g4` is the text file containing your grammar. Multiple examples can be found on the internet. **You will have to write your own grammar as part of the assignment!** Using the `-Dlanguage` flag, the target language can be chosen (*i.e.,* Python2 or Python3). Using the `-visitor` flag, a default parse tree visitor is generated.

Never modify the files generated by ANTLR, always write code in your own files and classes!

In order to manipulate the generated parser in Python, bindings must be installed. The reason for this is that the generated classes themselves depend on the `antlr4` library. This can be done automatically by executing the `pip` command (or an alternative package manager):

```
pip install antlr4-python3-runtime
```

Alternatively (if you do not have installation rights), the source code of the bindings can be downloaded from: `https://pypi.python.org/pypi/antlr4-python3-runtime`. Place the subfolder `antlr4` in your Python path.

A quick introduction on the Python bindings can be found here: `https://github.com/antlr/antlr4/blob/master/doc/python-target.md`.

See also Appendix A for more details on how to use ANTLR and some practical steps for the assignment.

2 Expression Parser

2.1 Grammar

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) Brackets to overwrite the order of operations.
- (mandatory) Logical operators `&&`, `||`, and `!`.

- (optional) Comparison operators `>=`, `<=`, and `!=`.
- (optional) Binary operator `%`.

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`").
- 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;
  ```

2.2 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 you do not depend on 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.

2.3 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.

Notes:

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

2.4 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.

Appendix: ANTLR Overview

The following steps are required to succesfully complete the assignment:

1. Create a grammar file (`.g4` extension). This file contains the grammar of the language you want to parse. Multiple examples can be found on the internet. **You will have to write your own grammar as part of the assignment!**

2. Generate the language parser in Python using the command:

   ```
   java -jar antlr-4.12.0-complete.jar -Dlanguage=Python3
   MyGrammar.g4 -visitor
   ```

   This will generate Python classes that you can use to parse files written according to your specific grammar. **Do not edit these files, as they will be overwritten everytime you change your grammar.**

3. Now you need to start writing Python code:

   (a) You will need a `main.py` that will handle the input and call the parser. You can find an example here: `https://github.com/antlr/antlr4/blob/master/doc/python-target.md`

   (b) You will need to create a subclass to the generated listener and visitor classes. The reason we use inheritance is to avoid losing your code when generating the ANTLR classes again. Write such classes in your own files.

   (c) Documentation for the ANTLR API can be found at `https://www.antlr3.org/api/Python/index.html`

   (d) Use the python `dir()` function with ANTLR objects as parameter to find out what fields and methods are available.

   (e) For more information on the API, you can use the documentation for the Java API: `http://www.antlr.org/api/Java/index.html`.

4. Create a simple text file with an example of your language (following the rules of your grammar). Run the `main.py` script and use this example file as input (via a command line argument).

Appendix: Project Overview

Reference

If you want to compare certain properties (output, performance, ...) of your compiler to an existing compiler, the reference is the GNU C Compiler with options `-ansi` and `-pedantic`. When in doubt over the behavior of a piece of code (syntax error, semantical error, correct code, etc.), GCC 4.6.2 is the reference. Apart from that, you can consult the ISO and IEC standards, although only with regards to the basic requirements.

Tools

The framework of your compiler is generated by specialized tools:

- In order to convert your grammar to parsing code, you use ANTLR. ANTLR has got several advantages compared to the more classic Lex/Yacc tools. On the one hand, your grammar specifications are shorter. On the other hand, the generated Python code is relatively readable.
- DO NOT edit generated files. Import and extend classes instead.

Make sure your compiler is platform independent. In other words, take care to avoid absolute file paths in your source code. Moreover, your compilation and test process should be controlled by the "test" script.

Deadlines and Evaluation

Evaluation:

- Make sure your compiler has been thoroughly tested on a number of C files. Describe briefly (in the README file) which input files test which constructions.
- You should be able to demonstrate that you understand the relations between the different rules.
- You should understand the role of a symbol table. Make sure you can indicate which data structure you use and how this relates to the AST structure.
- Show that every rule instantiates an AST class.
- Show which rules fill the symbol table and which rules read from it.

Deadlines: The following deadlines are strict:

- By **Friday 24 February 2023**, you should send an e-mail with the members of your group (usually 2 people, recommended).
- By **Friday 31 March 2023**, you should be able to demonstrate that your compiler is capable of compiling a small subset of C to the intermediary LLVM. This will be defined in project assignments 1 - 3.
- By **Friday 28 April 2023**, you should be able to demonstrate that your compiler is capable of compiling C to the intermediary LLVM (project assignments 1 - 6).
- By **Tuesday 30 May 2023**, the final version of your project should be submitted. The semantical analysis should be complete now, and code generation to both LLVM and MIPS should be working. Indicate, in the README file, which optional requirements you chose to implement.

No solutions will be accepted via e-mail; only timely submissions posted on BlackBoard will be accepted and assessed.

Reporting

At each evaluation, a version of your compiler should be submitted. Upload a zip file on Blackboard which contains the following:

- A minimal report that discusses your progress, discussing the implementation status of every required, and optional (if implemented), feature.
- ANTLR grammar.
- Python sources of the compiler.
- “build” and “test” scripts that can easily be used to demonstrate the functionality of your compiler.
- (if necessary) C example sources for the implemented functionality as well as the expected output for the respective examples.

2.5 Exam

The schedule for the final presentations will be available on Blackboard and discussed with all groups. In case you wish to report on the progress of your compiler at an earlier date than indicated, please let us know.