Compilers

Project 1: Expressions

2nd - 3rd Bachelor Computer Science 2019-2020

Brent van Bladel
brent.vanbladel@uantwerpen.be

For the project of the Compilers-course you will develop, in groups of 2 students (or alone, though this will make the assignment obviously more challenging), a compiler capable of translating a program written in a subset of C towards MIPS instructions. The compiler must be written in Python. From the large gamma of parser/AST generators you will use the Java-based ANTLR tool. This tool converts a declarative lexer and parser specification into Python code capable of constructing an explicit abstract syntax tree (AST) from a given C source file. The declarative parser specification consists of a grammar of the source language (C). The tree should be traversed a number of times and Python code should be added in order to check input programs for semantic validity, apply optimizations and generate MIPS code. You will do this over the course of the semester with weekly incremental assignments. 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

ANTLR (`http://www.antlr.org`) can be used out-of-the-box as a Java jar package. The latest version can be downloaded from `https://www.antlr.org/download/antlr-4.8-complete.jar`. It requires Java to run.

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

```
java -jar antlr-4.8-complete.jar -Dlanguage=Python2 MyGrammar.g4 -visitor
```

`MyGrammar.g4` is the text file containing your grammar. Multiple examples can be found here: `https://pragprog.com/titles/tpantlr2/source_code`. 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.

In order to manipulate the generated parser in Python, bindings must be installed. This can be done either automatically by executing the pip command:

```
pip install antlr4-python2-runtime
```

Or in case of Python 3:

```
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-python2-runtime
```

```
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.

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.
- (optional) Binary operator `%`.
- (optional) Comparison operators `>=`, `<=`, and `!=`.
- (optional) Logical 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`").
- 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.
- 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 are not dependend 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 dot format. This way it can be visualized by Graphviz. For a reference on the dot format, see `http://www.graphviz.org/content/dot-language`.

Notes:

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

2.4 Optional: 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 A: 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 here: `https://pragprog.com/titles/tpantlr2/source_code`.

2. Generate the language parser in python using the command:
```
java -jar antlr-4.8-complete.jar -Dlanguage=Python2 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 inheritence is to avoid losing your code when generating the python classes again.

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 command line argument).

Use the python `dir()` function with ANTLR objects as parameter to find out what fields and methods are available, and use the Python API: `http://msdl.cs.mcgill.ca/people/bart/compilers/antlr4-python2-doxygen.zip` (open index.html). For more information on the API, you can use the Java API: `http://www.antlr.org/api/Java/index.html`.

Appendix B: 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, ...), 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:

- To fully test each group member's understanding of their compiler, evaluations will take place **individually**.
- 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 **21 Februari 2020**, you should send an e-mail with the members of your group (usually 2 people, recommended).
- By **20 March 2020**, 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 **20 April 2020**, you should be able to demonstrate that your compiler is capable of compiling C to the intermediary LLVM.
- By **2 June 2020**, 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 point, a version of your compiler should be submitted. Upload a zip file on blackboard which contains the following (if applicable):

- 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.
- C sample sources, and oracles (if necessary).

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.