Compilers

Project 5: Functions

2nd Bachelor Computer Science 2021-2022

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 extend your compiler to support functions.

1 Functions

1.1 Grammar

Extend your grammar to support the following features:

- (mandatory) Reserved words.
  Minimally *return* and *void* must be supported, next to the previously supported reserved words.
- (mandatory) Scopes.
  Your compiler should support function scopes, next to the previously supported scopes from loops and conditionals.
- (mandatory) Local and global variables.

- (mandatory) Functions.
  This feature entails the definition and calling of functions and procedures, including support for passing parameters of basic types by value as well as by reference (pointers).

Example inputfile:

```
int mul(int x, int y){
        return x * y;
}

/*
 * My program
 */
int main(){
    int x = 1;
    while (x < 10) {
        int result = mul(x, 2);
        if ( x > 5) {
            result = mul(result, x);
        }
        printf(result); //show the result
        x = x + 1;
    }
    return 0;
}
```

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 you are not dependend on ANTLR classes.

1.3 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`.

1.4 Semantic Analysis

Extend your symbol table to support function scopes. Moreover, consistency of a return statement with the type of the enclosing function must be checked. You should also verify consistency between forward declarations and function definition signatures.

- (optional) Check whether all paths in a function body end with a return statement (not required for procedures that return void).

1.5 Optimizations

Extend your optimization visitor to support the following optimizations:

- (mandatory) Unreachable code and dead code.
  Do not generate code for statements that appear after a *return* in a function.
  - (mandatory) If you support the *break* or *continue* keywords, do not generate code for statements that appear after these keywords in a loop.
  - (optional) Do not generate code for variables that are not used.
  - (optional) Do not generate code for conditionals that are never true.

1.6 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`

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, ...), 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 **25 February 2021**, you should send an e-mail with the members of your group (usually 2 people, recommended).
- By **18 March 2022**, 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 **22 April 2022**, you should be able to demonstrate that your compiler is capable of compiling C to the intermediary LLVM (project assignments 1 - 6).
- By **27 May 2022**, 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).

1.7 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.