Compilers

Project 2: Variables

2nd Bachelor Computer Science 2021-2022

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 extend your parser to support variables.

1 Variables

1.1 Grammar

Extend your grammar to support the following features:

- (mandatory) Types.
  There should be support for the primitive data types `char`, `float`, `int`, and pointer types. The types `char`, `int`, and `float` become reserved words. Literals are now no longer limited to integers: literals of any type (integer, floating point) can now be part of expressions.

- (mandatory) Reserved words.

The `const` keyword must be supported, next to the types `char`, `int`, and `float`.

- (mandatory) Variables.
  There should be support for variables. This includes variable declarations, variable definitions, assignment statements, and identifiers appearing in expressions. You must also provide support for `const` variables.
  Note: A variable name (identifier) can contain only letters (both uppercase and lowercase letters), digits and underscore. The first character of an identifier should be either a letter or an underscore. There is no rule on how long an identifier can be.

- (mandatory) Pointer Operations.
  Support for the unary operators `*` and `&`.

- (optional) Increment/Decrement Operations.
  Support for the unary operators `++` and `--`.

- (optional) Conversions. Consider the following order on the basic types:
  `float isRicherThan int isRicherThan char`
  Implicit conversions of a richer to a poorer type (e.g. assignment of an `int` to a `char` variable) should cause a warning indicating possible loss of information. Another extension could be support for explicit casts (i.e. the cast operator). This enables the programmer to indicate he is aware of possible information loss. Hence the compiler should not yield a warning anymore.

Example inputfile:

```
const int x = 5*(3/10 + 9/10);
float y = x*2/( 2+1 * 2/3 +x) +8 * (8/4);
y = x + y;
int z;
float* flt_ptr = &y;
char ch = 'x';
```

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, so that your code files are separate from the ANTLR generated files.

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.

1.4 Constant Propagation

When a variable is `const` or used immediatly after being assigned, it can be evaluated at compile time. Hence, most compilers will not actually generate machine code (assembler) for the variable lookup in this situation. Rather, they will replace the variable node in the AST with a literal node containing the result.

Extend your optimization visitor to first replace identifiers in expressions with their value, if it is known at compile-time, before performing constant folding. Note that you should support constant folding if you want to support constant propagation (see project 1).

2 Error Analysis

2.1 Syntax Errors

The compiler is allowed to stop when it encounters a syntax error. An indication of the location of the syntax error should be displayed, but diagnostic information about the type of error is optional (and non-trivial).

2.2 Semantic Errors

For semantical errors, it is necessary to output more specific information about the encountered error. For example, for usage of a variable of the wrong type, you might output:

```
[ Error ] line 54, position 13: variable x has type y while it should be z
```

When in doubt, the GNU C Compiler with options `-ansi` and `-pedantic` is the reference.

Implement a semantic analysis visitor for your AST that checks for semantic errors. These errors include, but are not limited to:

- Use of an undefined or uninitialized variable.
- Redeclaration or redefinition of an existing variable.
- Operations or assignments of incompatible types.
- Assignment to an rvalue.
- Assignment to a `const` variable.

The semantic analysis visitor will make use of a *symbol table*. Define your own datastructure in Python to construct the symbol table. Keep in mind that the symbol table also needs to consider scopes, even though the current inputfiles only have a global scope.

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.

2.3 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.