Compilers

Project overview

2nd Bachelor Computer Science
2022 - 2023

Kasper Engelen
kasper.engelen@uantwerpen.be

1 Introduction

In this document you will find practical information concerning the project that you will work on throughout the semester. The project will consist of implementing a compiler that compiles code written in (a subset of) the C language. The compiler will output LLVM IR code and MIPS assembly code. The project will be completed over the semester with weekly incremental assignments. During the project you will work in groups of two students. You can also work alone, but this will make the assignment obviously more challenging.

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.

2 Reference Compiler

If you want to compare certain properties (output, performance, etc.) of your compiler to a real-world compiler, the reference is the GNU C Compiler with options `-ansi` and `-pedantic`. The `-ansi` and `-pedantic` flags will tell GCC to **strictly** adhere to the C89 standard. Compiling a C source code file called `test.c` using GCC can be done as follows:

```
gcc -ansi -pedantic test.c
```

You can then study the behaviour of the compiler: warnings, errors, behaviour of the compiled code, etc.

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.

3 LLVM Reference

For a reference on the LLVM intermediate representation, it is best to consider the official documentation: `https://llvm.org/docs/LangRef.html`. On StackOverflow you may find more specific examples and explanations.

Alternatively, you can use Clang to output LLVM IR code for a given C file:

```
clang -S -emit-llvm test.c
```

Clang can also be accessed using an online interactive tool called "Compiler Explorer": `https://godbolt.org/`. In order to get it to output LLVM IR for the C language, you can use the following options:

- Language: C
- Compiler: `x86-64 clang 16.0.0`
- Compiler options: `-S -emit-llvm`

4 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.

5 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 language. This will be defined in project assignments 1 – 3.
- By **Friday 5 May 2023**, you should be able to demonstrate that your compiler is capable of compiling C to the intermediary LLVM language (project assignments 1 – 6).
- By **Tuesday 30 May 2023**, the final version of your project should be submitted. The semantic 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.

6 Instructions for the evaluation for parts 1 – 3

Hand in a `zip` file on Blackboard that contains the following:

- a **README** file,
- your **grammar**,
- your Python **code**,
- **C files** that demonstrate the functionality, both **mandatory** and **optional**,
- a **test script that automatically runs your compiler** on the specified C files and, for each C file, produces two things:
  - the **AST** tree in Graphviz dot format,
  - the **LLVM** IR code,
- files that describe the **expected output** of your compiler.

Submissions that do not follow the above guidelines cannot be evaluated!

7 Instructions for the evaluation for parts 1 – 6 (LLVM)

The deliverable consists of two parts:

1. Hand in a `zip` file on Blackboard that contains the following:
   - a **README** file,
   - your **grammar**,
   - your Python **code**,
   - **C files** that demonstrate the functionality, both **mandatory** and **optional**,
   - a **test script that automatically runs your compiler** on the specified C files and, for each C file, produces two things:
     - the **AST** tree in Graphviz dot format,
     - some visualisation of the **symbol table**,
     - the **LLVM** IR code.
   - a link to the **video** that you made.
2. Create a **video** in which you demonstrate your compiler:
   - Upload this video to YouTube or another online video platform. You may also use Mediasite. Make sure the link to your video works and **does not require an account/login.**
   - **Do not upload the video to Blackboard!**
   - The video should be **10 minutes long.**
   - Demonstrate that your compiler is capable of compiling to **LLVM IR**.
   - Show the **AST** and the **symbol table**.
   - Show that the compiler works on the **test files on Blackboard**.
   - Make sure that your **optional functionality** is also demonstrated. The manda-

tory functionality is sufficient for a grade of 10/20. The optional functionality goes towards grades 11/20 – 20/20. Optional functionality will only be considered if the mandatory functionality is present.

Submissions that do not follow the above guidelines cannot be evaluated!

8 Grading

The project counts for 60% of the entire course (=12/20). The LLVM part counts for 40% of the project (=4.8/20), the MIPS part counts for the other 60% (=7.2/20). Both parts of the project are graded separately.

If you implement **all mandatory functionality**, then you get 50% of the points for that part of the project. The amount of implemented optional functionality is then used to determine a grade between 50% and 100%.

We first take a look at how many mandatory features you implemented, by counting the number of implemented features and dividing by the total amount of features, so that the total grade is:

$$\frac{\text{number of implemented mandatory features}}{\text{total number of mandatory features}} \times 0.5 \times \text{weight of that part of the project}$$

If you implemented all mandatory features, we also count the number of optional features in a similar way, and then the following is added to your grade:

$$\frac{\text{number of implemented optional features}}{\text{total number of optional features}} \times 0.5 \times \text{weight of that part of the project.}$$

Note that you can also come up with your own optional features, which will also count towards a higher grade.

Example: You implemented the following:

- 8 out of 30 mandatory features for LLVM,
- 2 out of 12 optional features for LLVM,
- 1 feature for LLVM that you invented/proposed yourself,
- 30 out of 30 mandatory features for MIPS,
- 5 out of 12 optional features for MIPS,

- 2 features for MIPS that you invented/proposed yourself.

Then your score is:

$$\left(\frac{8}{30} \times 0.5 + \frac{0}{12} \times 0.5\right) \times \frac{4.8}{20} + \left(\frac{30}{30} \times 0.5 + \frac{5+2}{12} \times 0.5\right) \times \frac{7.2}{20}$$
$$= \frac{6.34}{20} = 53\% \text{ of the grade for the project}$$

9 Summary of functionality

Below you will find a summary of all functionality that will be considered during grading. Note that this list is only a summary, the individual assignment PDFs are the reference in case of doubt. For a more detailed explanation, please see the assignment PDF.

9.1 Mandatory

The following list comprises the mandatory functionality, make sure that you have implemented all the following items.

- Assignment 1:
  - Binary arithmetic operators: `+`, `-`, `*`, `/`
  - Binary comparison operators: `<`, `>`, `==`
  - Unary operators: `+var`, `-var`
  - Parentheses, order of operation
  - Logical operators: `&&`, `||`, `!var`
  - Ignore whitespace in code
  - Constant folding
  - Constructing and using the AST
- Assignment 2:
  - Types (`float`, `char`, `int`)
  - Variables
  - Constant variables
  - Pointers
  - Basic pointer operations: address (`&var`), dereference (`*var`)
  - Constant propagation
  - Construction and usage of the symbol table
  - Syntax errors + error message
  - Semantic errors + error message:
    - Usage of uninitialised and undeclared variables

        - Redeclarations and redefinitions of existing variables
        - Operations and assignments with incompatible types
        - Assignment to an rvalue expression
        - Re-assignment of const variables
- Assignment 3:
    - Single line and multi-line comments
    - (the `printf` from assignment 3 is temporary, you should not support it after assignment 6)
- Assignment 4:
    - `if` and `else` statements
    - `while` loops
    - `for` loops
    - `break` statements
    - `continue` statements
    - Anonymous scopes
    - Symbol table for anonymous scopes
- Assignment 5:
    - Function scopes
    - Symbol table for function scopes
    - Local and global variables
    - Functions:
        - Definition and declaration
        - Calling
        - Parameters (primitives and pointers, pass-by-value, pass-by-reference)
        - Return values
        - Functions with `void` return
    - Type checking for `return` statements
    - Type checking for forward declarations
    - No dead code after `return`, `break`, and `continue`
- Assignment 6:
    - One-dimensional static arrays
    - `printf`
    - `scanf`
    - `#include <stdio.h>` statement (only for `printf` and `scanf`)

9.2 Optional

The list below contains a number of optional functionalities that we propose. You may also come up with your own ideas for additional functionalities.

- Assignment 1:

  - Additional logical operators: `>=`, `<=`, `!=`
  - Modulo operator: `%`
- Assignment 2:
  - Increment, decrement operators: `++`, `--` (both prefix and suffix variants)
  - Type conversions (implicit and explicit)
- Assignment 3:
  - Store comments in AST and machine code
  - Comment machine instructions with the original C statement
- Assignment 4:
  - `switch` statements with `case`, `break`, and `default`
- Assignment 5:
  - Check that all execution paths end in a `return` statement
  - No code is generated for unused variables
  - No code is generated for if-clauses that are never true
- Assignment 6:
  - Multi-dimensional arrays
  - Assignment of complete rows to array indices
  - Dynamic arrays