Compilers

Project overview

2nd Bachelor Computer Science
2023 – 2024

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.

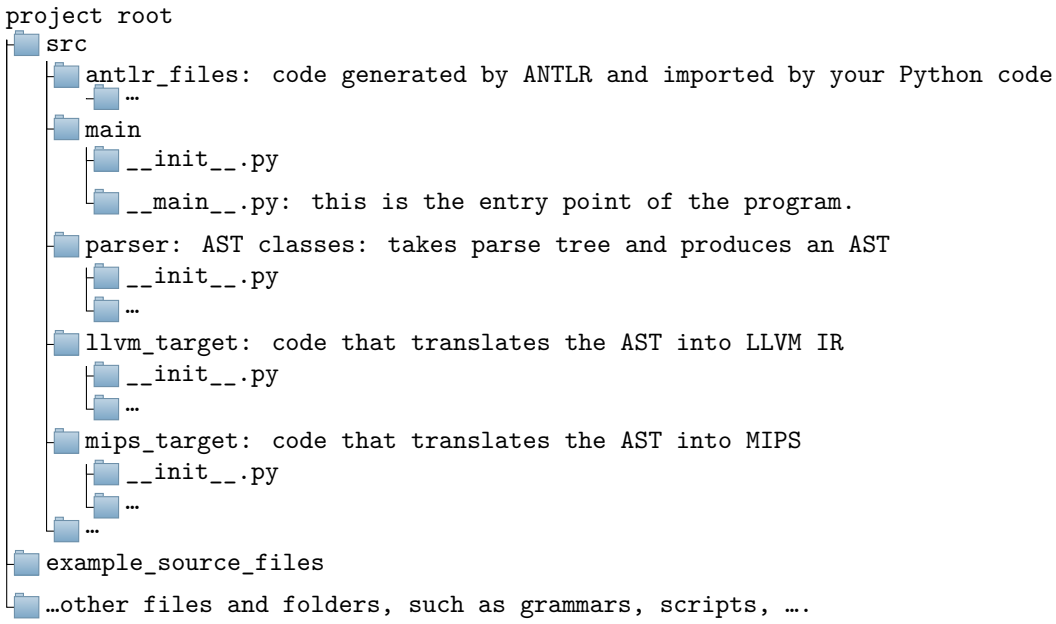
Concretely, you will construct a grammar and lexer specification, which is then turned into Python code using the ANTLR tool. The rest of the project will consist of developing custom code written in Python 3. This custom code will take the parse tree and turn it into an abstract syntax tree (AST). This 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 ANTLR

In this project we will use the ANTLR tool. On Blackboard you can find a PDF with additional information on ANTLR.

3 Project structure

In order to make sure that the assistant can read your code, give feedback, offer advice, and **grade your project**. it is important that a proper Python project structure is maintained. For example:



Invoking the compiler: In order to parse arguments, you can make use of the built-in `argparse` library. Your compiler can be accessed by issuing the following commands from the **project root directory**:

- Rendering the AST:

```
python -m src.main --input input_file.c --render_ast ast_output.dot
```

- Rendering the symbol table:

```
python -m src.main --input input_file.c --render_symb symb_output.dot
```

- Compile to LLVM:

```
python -m src.main --input input_file.c --target_llvm output_file.ll
```

- Compile to MIPS:

```
python -m src.main --input input_file.c --target_mips output_file.mips
```

Make sure your compiler is platform independent. In other words, take care to avoid absolute file paths in your source code.

4 Submitting your solution

In order to submit your solution, you will have to closely follow the steps below. Normally speaking, these instructions are final, but additional instructions might be issued via **Blackboard announcements**.

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**,
     - if applicable, the **LLVM IR** code,
     - if applicable, the **MIPS** 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.**
3. The video should have the following "script":

(Step 1) Explicitly go over the **requirement list on Blackboard**, indicating for every item whether or not this is implemented.

(Step 2) Demonstrate that your compiler is capable of compiling to **LLVM IR** or **MIPS**, depending on the assignment, by compiling the **test files on Blackboard**.

(Step 3) Show the **AST** and the **symbol table** for some interesting examples.

(Step 4) Make sure that your **optional functionality** is also demonstrated. The mandatory 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.

5 Note on groups of 4 students

The groups that have 4 students instead of three have to fullfill the requirements below, on top of the normal requirements for the project:

- The following optional features are mandatory for groups of 4, in addition to the other mandatory features:
  - Include guards (assignment 5)
  - Unions (assignment 7)
  - Function pointers (assignment 7)
- Aside from presenting their solution in a video, groups of 4 will have to give a **presentation** where they demonstrate their compiler. Every member of the group is expected to **thoroughly understand** the internal workings of the compiler.
- All members of the group are expected to make an **equal contribution** to the project.
- The presentation, submitted compiler, etc. are **all taken into account** while calculating the final grade.

6 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.

7 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`

8 MIPS Reference

There are a number of useful sources regarding the syntax and semantics of MIPS:

- the **CSA course** of the 1st bachelor year,
- various MIPS **reference sheets**,
- **Stackoverflow** for specific questions and problems,
- the **auto-complete** feature of the MARS simulator,
- a list of **system calls** and their respective arguments: Link,
- *Computer Organization and Design, MIPS Edition* by David Patterson and John Hennessy.

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

- Language: C
- Compiler: `mips clang 16.0.0` (GCC works also, but does not use the proper register names)
- Compiler options: leave empty

Note: The output of Clang does not fully correspond with the MIPS code your compiler needs to generate:

- The code generated by "real" compilers may not always work on your emulator, since emulators do not always provide the full functionality of the MIPS instruction set. In the case that happens, you'll have to manually correct the MIPS code generated by Clang to make it work in MARS or SPIM.
- Also note that Clang does not generate an `exit` syscall at the end of the `main` function, so you'll have to add this yourself.
- You'll manually have to implement `printf` and `scanf`. Clang will sometimes just do `jal printf`, which is not sufficient for this project.

9 Test files

On Blackboard you will find examples of C-code. You can use these to test your compiler and to demonstrate during the evaluation video that your compiler works as intended. Note, however, that these files are **not exhaustive**. They are provided to help you find bugs in your compiler.

It is up to you to properly test the compiler, and you are expected to implement all required functionalities, as well as the optional functionalities that you indicated. If certain functionalities are not properly implemented, points will be deducted.

The test files for the first three assignments have the following filename structure:

```
<proj1/proj2/proj3>_<man/opt>_<pass/syntaxErr/semanticErr>_<explanation>.c
```

- `<proj1/proj2/proj3>`: project 1, 2, or 3
- `<man/opt>`: mandatory or optional functionality
- `<pass/syntaxErr/semanticErr>`: the expected result: success, syntax error, semantic error.
- `<explanation>`: some explanation

10 Code generation

Please pay attention to the following:

- All features should be implemented for both LLVM IR and MIPS by the end of this project. Features that are only implemented for one of the two targets do not count.
- You should use a library to generate LLVM IR, such as llvmlite (Example). This will allow you to build your LLVM code in an object-oriented manner.

11 Deadlines

The following deadlines are strict:

- By **Friday 23 February 2024**, you should sign up for one of the groups **on Blackboard**.
- By **Friday 29 March 2024**, 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 3 May 2024**, you should be able to demonstrate that your compiler is capable of compiling C to the intermediary LLVM language (all assignments).
- By **Tuesday 28 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.

12 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. There is no re-take (tweede zit) for the project.

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:

$$\begin{aligned}\left(\frac{8}{30} \times 0.5 + \frac{0}{12} \times 0.5\right) \times \frac{4.8}{20} + \left(\frac{30}{30} \times 0.5 + \frac{5+2}{12} \times 0.5\right) \times \frac{7.2}{20} \\ = \frac{6.34}{20} = 31.7\%\end{aligned}$$