Objectives

To investigate appropriate tools to assist with the implementation of the code generation phase of the compiler. To participate in the peer evaluation activity to assess and to aid with the progress of the back-end compiler development. To develop a fully documented and tested code generator for the chosen source and target languages, implemented in the programming language of the developer’s choosing.

Reading Assignment

If you have not done so already, please read the “Mastering Markdown” and “Documenting Your Projects on GitHub” GitHub Guides available at https://guides.github.com/. To have a better understanding of the code generator concepts, you should also read Chapters 7 and 8 in the BCD course textbook.

Code Generator

In this final lab, you will implement a back-end for your compiler that will generate code. Finally, you get to the true product of all your labor-running source programs!

This pass of your compiler should traverse the syntax tree, stringing together the appropriate TAC instructions for each subtree to assign a variable, call a function, or whatever is needed. Those TAC instructions are then translated into an assembly language of your choice. You may also decide instead to skip conversion to TAC, and generate assembly code directly from the syntax tree. Your finished compiler will do code generation for some of your source language as well as reporting errors.

The debugging can be intense at times since you may need to drop down and examine the assembly code to sort out the errors. By the time you are done, you will have a pretty thorough understanding of the runtime environment for your source programs and will gain a little more reading familiarity with the assembly. The back-end of the compiler is where the systems-specific portion of the work comes in. I think this is definitely the most fun of the projects; it’s a great feeling when you finally get to the stage where you can compile source programs and execute them.

You may choose any tools necessary to complete this portion of the compiler, however it is likely that you will have to write significant portion of the code yourself. You should then test your completed compiler’s output using varying input files. For example, to execute the assembly and to test your output, you can use GNU assembler.

Required Implementation

Although, you are encouraged to implement as much of your source language’s functionality as possible, you are only required to provide a correct implementation for the following base requirements:

- arithmetic operations (+,-,*) with constants and variables
• data types (at least two)
• functions
• conditional statements (at least one)
• loops (at least one)
• printing
• comments

I will consider significant implementation beyond these baseline requirements for extra credit.

**Grading and Required Deliverables**

Specifically, this assignment invites you to submit, using GitHub, the following deliverables. You are to use the same repository from your previous lab as it is to contain the implementation of your full compiler at the end of the semester.

• Fully documented programs with the code generation implementation that are free of errors. Your program(s) should be stored in the `src/` directory of your repository. It should be clear from your directory structure and file naming conventions what constitutes your back-end.

• A comprehensive documentation on the description of the tools and languages necessary for your implementations and the detailed steps to follow to build/run your complete compiler. This information should be written in the Markdown language and be added to your repository’s `README` file.

• Submit a `reflection.md` file that explains your overall compiler design decisions. Describe how your code is structured for the back-end, and explain why you believe the design is a good one (i.e. why it leads to a correct code generator). Also, comment on overall challenges you have encountered throughout the semester with this compiler development project and how you tackled those challenges.

• Small test files used to test your completed compiler, include examples of input programs that correctly compile and execute your source language into your target language. You must have at least ten small input program(s) that are stored in the `tests/` directory of your repository and that are responsible for testing different functionality of your compiler.

• Output can be printed to the terminal or to the file. If the input is not scanned or parsed correctly, you must produce a reporting of the error. All of the scanning and parsing errors possible for your grammar should be caught at this point. If you are not doing error reporting at the scanning/parsing level for some part of your grammar you should either modify your grammar or explain what is not supported in your documentation.

• Give a demonstration of your completed compiler in class on April 25.

Points for this lab will be distributed as follows:

• Design, Correctness and Completeness: 70%
• Documentation: 15%
• Demonstration: 5%
• Peer Review: 10%

Most of the points are allocated for correctness and completeness of the base requirements, but I will also evaluate your design decisions (as described in your README and your reflection) and the cleanliness and efficacy of your implementation. I will grade your solution manually by running the programs and by looking through your implementation, so please comment your code well!

Once you have finished this lab assignment, you will have built a complete working compiler in a fast-paced 12 weeks. Congratulations - this is no mean feat!

**Peer Evaluation**

During the lab session on **April 16** we will have a peer review session. The goal of this session is to assess overall design of back-end compiler, to catch any big red flags, and to aid with the completion of your implementation. During this session each developer will be partnered with a peer developer to explain your overall code generation design and to complete a code walkthrough of your selected completed implementation. Each reviewer will complete a peer review summary, which will be provided to the class beforehand.

Before the peer evaluation session please ensure that you have made a tremendous progress towards your implementation. You must have some code by this time that can be reviewed and you should demonstrate simple input test files. It is not necessary to have completed test cases and a fully implemented parser by the peer evaluation session.

**Recommended Lab Completion Timeline**

The following timeline is recommended to ensure you are able to complete this lab by the due date.

- **Week 1** (April 2 - April 5): Revisit error handling in the parser. Make your implementation design. Prepare for the implementation.
- **Week 2** (April 8 - April 12): Begin implementation.
- **Week 3** (April 15 - April 19): Continue implementation and begin testing. You should have some preliminary implementation and a couple of test cases to show during the peer review session.
- **Week 3** (April 22 - April 25): Complete implementation and testing.

**Adhering to the Honor Code**

In adherence to the Honor Code, students should complete this assignment on an individual basis. While it is appropriate for students in this class to have high-level conversations about the assignment, it is necessary to distinguish carefully between the student who discusses the principles underlying a problem with others and the student who produces assignments that are identical to, or merely variations on, someone else’s work. Additionally, since there are many compiler implementations available online, please mention any outside tools and projects you have used to help you. Deliverables that are nearly identical to the work of others will be taken as evidence of violating the Honor Code. Please see the course instructor if you have questions about this policy.