Esther Fatehi, fatehie@oregonstate.edu
Josh Huff, huffj@oregonstate.edu
Jose Murrieta, murrietj@oregonstate.edu
Caelum - CMP1
Instructor Brewster
CS 467 – Spring 2018
April 15, 2018

Capstone Project Plan: COOL Compiler

Introduction

The Caelum Team will be writing a compiler that takes a program written in COOL (Classroom Object Oriented Language) and translates it into assembly source code. Until now, the process of compilation has been abstracted away as a command to be run on a CLI or a button to be pressed on an IDE. Because OSU's post-bacc program does not offer a course on compiler design, this is each team member's first exposure to the essential concepts of compilers beyond that abstraction, as well as the collective first exposure to COOL and MIPS assembly. Consequently, the free, self-paced Stanford compiler course will be the source of the bulk of our knowledge and direction.

Expected User Experience

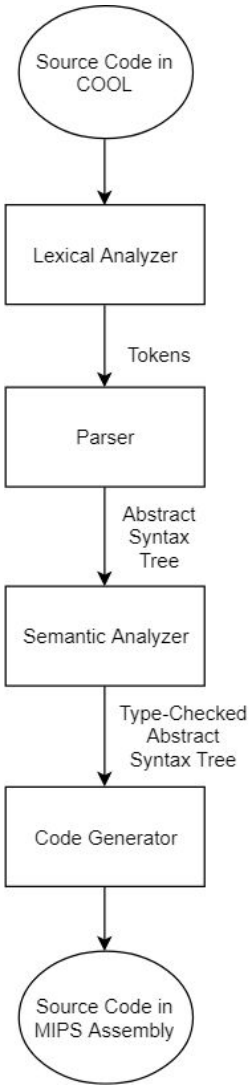
The target user for this application is a software developer with experience in compiled languages and familiarity with basic use of the command line interface.

From the user's perspective, the end result of our work will be functionally indistinguishable from any other CLI-based compiler with which they have experience. That is, a user able to write source code in COOL can run it through our compiler. Assuming there are no syntax errors in the given file, the compiler will translate the static COOL statements into a working equivalent in the target language, in this case, MIPS assembly code.

Initial Thoughts on Structure

Though modern compilers are being produced with extra features, the principles underlying their construction seem relatively untouched. After all, the “Dragon book” has only seen one major revision in 30 years.

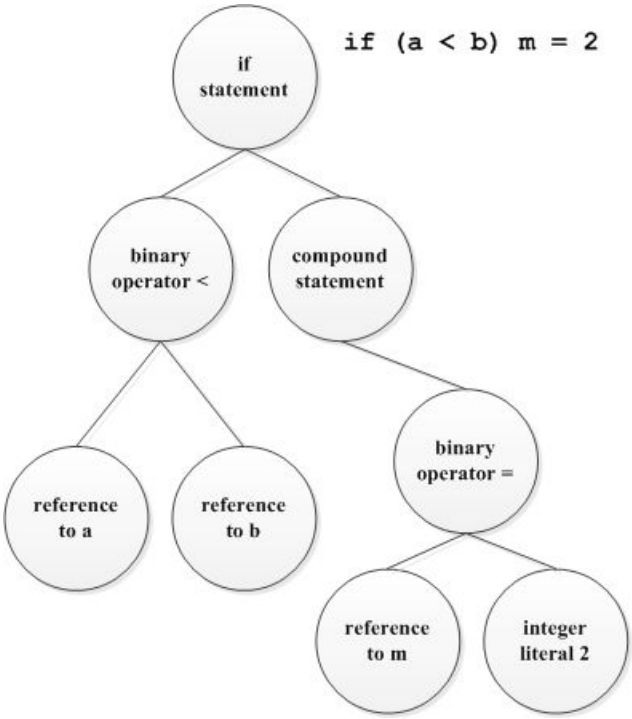
Figure 1: Anatomy of a Compiler



As demonstrated in Figure 1, the COOL source file will be the input for the lexer, which will do lexical analysis by tokenizing each element. The tokens will then go into the parser, which will create a syntax tree. The syntax tree goes into a translator which does semantic analysis to produce a type-checked syntax tree, which is interpreted by the code generator to produce assembly code. The sequential nature of the program

lends itself to following the imperative paradigm and dividing the discrete phases of operation into modules. The group has unanimously decided that the compiler will be written in C++.

Figure 2: Abstract Syntax Tree (AST) Example



Source: https://www.ics.com/blog/introduction-clang-part-2

An important data structure in our compiler is the Abstract Syntax Tree (AST). The AST is used to represent the syntactic structure of the source code. It is the job of the parser to produce this from the tokens that it receives from the lexical analyzer. It is called an *Abstract* Syntax Tree because many of the unnecessary details, such as punctuation and delimiters, are omitted to simplify the compiler's job. Figure 2 above shows an example of what an AST would look like for a snippet of C code.

Necessary Technologies and Knowledge

At present, none of us is a subject matter expert. Therefore, we expect to discover more about the project's required languages and technologies in the initial weeks, which we've allocated as a research phase.

Due to the low-level nature of this project, there is little in the way of dependencies as there would be in the design of a web app or video game. Without a GUI or front-end experience to engineer, we have no need to leverage the functionality provided by frameworks, libraries, and APIs.

Instead, our success will rely on the use of VMware to mimic the architecture and otherwise allow for our compiler to operate without concern for peculiarities in environment or discrepancies in platform. Assembling the result of compilation will require Spim, which is a simulation of the MIPS processor.

Team Member Duties

Esther Fatehi

Develop and unit test the Code Generation module
Develop and unit test the Optimizer module (optional and schedule-permitting)
Supply information for final report

Josh Huff

Develop and unit test the Semantic Analyzer module
Develop and unit test the Optimizer module (optional and schedule-permitting)
Complete final report

Jose Murrieta

Develop and unit test the Lexical Analyzer module
Develop and unit test the Parser module
Supply information for final report

All Team Members

Study all Stanford Online "Compiler" course lectures and COOL documentation
Integrate the completed and unit-tested components
Complete integration testing individually to ensure maximum test coverage

Tentative Development Schedule

	Esther Fatehi	Josh Huff	Jose Murrieta
Week 3	Watch lectures (10 hrs)	Watch lectures (10 hrs)	Watch lectures (10 hrs)
Week 4	Watch lectures (10 hrs)	Watch lectures (10 hrs)	Watch lectures (10 hrs)
Week 5	Pseudocode and Planning (10 hrs)	Pseudocode and Planning (10 hrs)	Pseudocode and Planning (10 hrs)
Week 6	“Hello World” for Mid-Point (15 hrs)	“Hello World” for Mid-Point (15 hrs)	“Hello World” for Mid-Point (15 hrs)
Week 7	Code Generator / Optimize* (15 hrs)	Semantic Analyzer / Optimize* (15 hrs)	Lexer / Parser (15 hrs)
Week 8	Unit Testing (10 hrs)	Unit Testing (10 hrs)	Unit Testing (10 hrs)
Week 9	Integration Testing (15 hrs)	Integration Testing (15 hrs)	Integration Testing (15 hrs)
Week 10	Delivery, Documentation, and Demonstration (10 hrs)	Delivery, Documentation, and Demonstration (10 hrs)	Delivery, Documentation, and Demonstration (10 hrs)
Total	115 hours	115 hours	115 hours

Conclusion

This project is, in a word, laborious. It is a front-loaded enterprise in that it will require a substantial degree of self-directed study for each member just to get up to speed with the theoretical aspects of the subject matter. Furthermore, it requires that the team become proficient enough with an obscure and unfamiliar OOP language to translate it into an equally obscure and unfamiliar assembly language. In addition to remembering long-lost concepts about computer architecture and assembly, we must develop a strong grasp of theories new to each of us and facilitate their practical implementation. Finally, this project will require us to communicate and coordinate in a way previous group projects have not. We rely on each other to complete our respective assignments, because one missing part of the structure means a failure for the whole.