CIS*4650 Project:
Milestone 3

Ben Carlson 1044277 carlsonb@uoguelph.ca

April 2022

Summary (TLDR)

This checkpoint/milestone involved the creation of code generation and simulation on all c minus code. The primary deliverables for this milestone are as follows:

1. Complete the C- compiler through assembly code generation
2. Debugging of symbol table + syntax tree
3. Gain familiarity with TMsimulator, the assembly code simulator for this milestone

Most of this milestone consisted of refactoring visitor classes and brushing up on assembly language. By doing so we were able to create a somewhat working product, with a few limitationns due to technical debt.

Code Generation Features

1. Declaration of variables
2. Variable assignment
3. input/output functions
4. Conditionals and loops (limited by call sequence bugs)

Limitations

1. Will generate assembly for all clean compiling files
2. Will not work on arrays
3. Call sequence is messed up, input() and output() calls need to be reversed

Techniques utilized/Overall Design
When creating the code generation, I aimed to work using SOLID and other OOP principles. Doing so aided in debugging later as the code created was very modular and easy to trace bugs with. We followed traditional OOP naming conventions, following camel case for most variable naming conventions. The overall design of the compiler is built off of two packages (absyn and symbol) followed by three driving classes (showtreevisitor, semanticanalyzer and asmgenerator). Other libraries utilized are JFlex (for scanning and tokenizing) and cup (used for parsing and some syntactical analysis). Cup serves as the point of entry for the ASM generator and semantic analyzer, which is called through the driving class main.java.

When looking at this milestone, similar to others I followed a 2-3 stage process with a debug at the end of the project.

Basic Understanding/Design
This assignment relies on a handful of changes to assembly code, such as three address as opposed to traditional ASM. The first part of this assignment I spent reading up on three-address and how it functioned similar to assembly. Following this, I then began to get familiar with the TMSimulator, the main testing point for the code generation. One final point of research I did was brushing up on assembly and what types of instructions I would need to use for this milestone. From here I could rough out what was needed to be done to get the MVP completed for this milestone.

Refactoring of core code
Going into this milestone, our code had definitely taken on some technical debt. Some of the code I made an effort to clean up and refactor, whereas other parts were too much to take on alone without my teammate. I cleaned up most of our driver class's code, from command line arg parsing to changing where files are generated (ie: moving file writing into cup as opposed to main). This also involved passing variables into the cup parser, shifting our program's point of entry primarily into CUP as opposed to main. Other issues arose during the ASM generation portion of this milestone, which will be discussed below.

ASMGenerator.java

After doing a refactor of the main class, I opted to move towards generating the actual assembly code. I started first by generating the important parts of the program, prelude and finale, and then moving towards more complex parts. Around the middle of testing the code generation, I found a very large bug in the design of our semantic analyzer. Our call sequence was out of order, making the order of operations within our programs occur in a semi reversed order. I still kept pushing on, leaving this design issue to the very end in the event I had time to revisit it. Ultimately, being a team of one for this assignment I was not able to revisit the call sequence problem. However, our assembly generation was able to work successfully on a handful of conditions (see below)

Debugging

At the end of the development process, I opted to debug the program. I tested loops, conditionals, arithmetic and variable assignment (array and non array). Ultimately this program ended up working in most basic cases, but more complex items (such as loops and arrays) either threw errors in the generator or in the assembly code itself. In addition to this, when testing input and output functions, I noticed that the call sequence within our semantic analyzer was reversed. This meant that functions such as input and output would be called out of order, due to assembly code being generated before the other parts.

Learning Experiences

- Learned about different modules of a compiler and their interaction
- Learned about code generation patterns
- Learned about the importance of cleaning up technical debt

Assumptions

Required software

1. JFlex
2. Java/JDK
3. CUP/JCUP
4. TMSimulator

Limitations

Supported files

1. .cm cminus files
2. Code gen only works on a small subset of files/features

Improvements

1. Top to bottom refactor/debug of semantic analyzer
2. More detailed/runtime error checking

3. Further code generation (arrays, fix call order)

Contributions

Ben

1. Created ASMGenerator.java
2. Refactored parts of semanticanalzyer + symbol table
3. Wrote code generation
4. Debugging
5. created documentation

Acknowledgements
This project was heavily based on/inspired by resources provided by Dr. Fei Song's compilers content. We were provided a rough idea of the implementation process as well as a portion of starter code. The grammars created in CUP are heavily based off the the C- specs. The absyn class structure is heavily based off of lecture content.