Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Spring 2021 Handout — Project Overview

This is an overview of the course project and how we'll grade it. You should not expect to understand all the technical terms, since we haven't yet covered them in class. We're handing it out today to give you some idea of the kind of project we're assigning, and to let you know the various due dates. Additional handouts will provide the technical details of the project.

The first project (Scanner and Parser) will be done individually. For subsequent projects, the class will be partitioned into groups of three or four students. You will be allowed to choose your own partners as much as possible. Each group will write a compiler for a simple programming language. We expect all groups to complete all phases successfully. The start of the class is very fast-paced: do not fall behind!

The Project Segments

Descriptions of the five parts of the compiler follow in the order that you will build them.

Scanner and Parser

A Scanner takes a Decaf source file as an input and scans it looking for *tokens*. A *token* can be an operator (ex: "*" or "["), a keyword (*if* or *class*), a literal (14 or 'c') a string ("abc") or an identifier. Non-tokens (such as white spaces or comments) are discarded. Bad tokens must be reported.

A Parser reads a stream of tokens and checks to make sure that they conform to the language specification. In order to pass this check, the input must have all the matching braces, semicolons, etc. Types, variable names and function names are not verified. The output can be either a user-generated structure or a simple parse-tree that then needs to be converted to a easier-to-process structure.

We will provide you with a grammar of the language, which you will need to separate into a scanner specification and a parser specification. While the grammar given should be pretty close to the final grammar you use, you will need to make some changes. You may use the approved parser generators to generate the scanner and parser.

Semantic Checker

This part checks that various non-context free constraints, e.g., type compatibility, are observed. We'll supply a complete list of the checks. It also builds a symbol table in which the type and location of each identifier is kept. The experience from past years suggests that many groups underestimate the time required to complete the static semantic checker, so you should pay special attention to this deadline.

It is important that you build the symbol table, since you won't be able to build the code generator without it. However, the completeness of the checking will not have a major impact on subsequent stages of the project. At the end of this project the front-end of your compiler is complete and you have designed the intermediate representation (IR) that will be used by the rest of the compiler.

Code Generation

In this assignment you will create a working compiler by generating unoptimized x86-64 assembly code from the intermediate format you generated in the previous assignment. Because you have relatively little time for this project you should concentrate on correctness and leave any optimization hacks out, no matter how simple.

The steps of code generation are as follows: first, the rich semantics of Decaf are broken-down into a simple intermediate representation. For example, constructs such as loops and conditionals are expanded to code segments with simple comparison and branch instructions. Next, the intermediate representation is matched with the Application Binary Interface, i.e., the calling convention and register usage. Then, the corresponding x86-64 machine code is generated. Finally, the code, data structures, and storage are laid-out in the assembly format. We will provide a description of the object language. The object code created using this interface will then be run to generate the required output.

Data Flow Analysis

This assignment phase consists of implementing data-flow optimizations, which usually involves setting up a data-flow framework. We will provide a description of the requirements of this phase in a later handout.

Optimizer

The final project is a substantial open-ended project. In this project your team's task is to generate optimized code for programs so that they will be correctly executed in the shortest possible time.

There are multitude of different optimizations you can implement to improve the generated code. You can perform data-flow optimizations such as constant propagation, common sub-expression elimination, copy propagation, loop invariant code motion, unreachable code elimination, dead code elimination and many others using the framework created in the previous segment. You can also implement instruction scheduling, register allocation, peephole optimizations and even parallelization across multiple cores of the target architecture.

In order to identify and prioritize optimizations, you will be provided with a benchmark suite of a few simple applications. Your task is to analyze these programs, perhaps hand optimizing them, to identify which optimizations will have the highest performance impact. Your write-up should clearly describe the process you went through to identify the optimizations you implemented and justify them.

This phase requires a project writeup describing the design of the group's chosen optimizations. This document will also count towards the project grade. Further information will be provided in a later handout.

Derby

The last class will be the "Compiler Derby" at which your group will compete against other groups to identify the compiler that produces the fastest code. The application used for the Derby will be provided to the groups a few days before the Derby. This is to give the group time to fix their

compiler in case it fails to correctly compile the Derby program. However, the group is forbidden from adding any application-specific hacks to make this specific program run faster.

Grading

Phases 2 and 3 of the project (Semantic Checking and Code Generation) will be graded as follows:

- (20%) Documentation. Your score will be based on the clarity of your documentation, and incisiveness of your discussion on design, possible alternative designs, and issues. Overall, a few pages for the supporting documentation is fine.
- (80%) Implementation (objective). Points will be awarded for passing specific test cases. Each project will include specific instructions for how your program should execute and what the output should be. If you have good reasons for doing something differently, consult the TAs first.

Phases 4 and 5 of the project (Data-flow Analysis and Optimization) will be graded differently:

- (20%) Documentation, with particular attention given to your description of the optimization selection process.
- (60%) Implementation. As each group implements different optimizations, the only public test is the generation of correct results for the benchmark suite and the Derby program (50%). The other half will be based on the design of the chosen optimizations and being able to implement at least one project 4 and one project 5 optimization.
- (20%) Derby Performance. The formula for translating the running time of the program compiled by your compiler into a grade will be announced later.

All members of a group will receive the same grade on each part of the project unless a problem arises, in which case you should contact your TA as soon as possible.

Documentation / Write-ups

All project writeups should be included in your project repository in the `doc/` directory. They should be clear, concise and readable. Fancy formatting is not necessary, just give us a clear idea what you have done. Acceptable file formats are pdf.

Your documentation must include the following parts, which could be described as Design, Extras, Difficulties, and Contribution. Not every question or point of each part need to be addressed, just enough information to describe each portion effectively:

1. **Design** - An overview of your design, an analysis of design alternatives you considered, and key design decisions. Be sure to document and justify all design decisions you make. Any decision accompanied by a convincing argument will be accepted. If you realize there are flaws or deficiencies in your design late in the implementation process, discuss those flaws and how you would have done things differently. Also include any changes you made to previous parts and why they were necessary. This section should aid the TA in being able to read and give feedback on the code written.

2. **Extras** - A list of any clarifications, assumptions, or additions to the problem assigned. This include any interesting debugging techniques/logging, additional build scripts, or approved libraries used in designing the compiler. The project specifications are fairly broad and leave many of the decisions to you. This is an aspect of real software engineering. If you think major clarifications are necessary, consult the TA.

3. **Difficulties** - A list of known problems with your project, and as much as you know about the cause. If your project fails a provided test case, but you are unable to fix the problem, describe your understanding of the problem. If you discover problems in your project in your own testing that you are unable to fix, but are not exposed by the provided test cases, describe the problem as specifically as possible and as much as you can about its cause. It is to your advantage to describe any known problems with your project; of course, it is even better to fix them. Also describe any section of your project that you would like to highlight for more feedback on/had questions on.

4. **Contribution** - A brief description of how your group divided the work. This will not affect your grade; it will be used to alert the TAs to any possible problems.