Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Spring 2023 | Handout — Project Overview | Monday, Feb 6

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. Additional handouts will provide the technical details of the project.

By the end of the course, you will have built a compiler that takes Decaf programs, a subset of C, as input and generates executable x86 code as output.

Organization. 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!

Deadlines. For up-to-date deadlines, refer to the course website.

1 Project Segments

Short, informal descriptions of the five parts of the compiler follow. These descriptions are intended as an informal overview of the project. The Decaf Spec and project handouts will provide the full technical details.

1.1 Scanner and Parser

A *scanner* takes a Decaf source file as an input and outputs a sequence of *tokens*, sequences of characters which form the basic units of Decaf programs. A token can be:

- an operator (e.g., `*` or `[`)
- a keyword (e.g., `if` or `while`)
- an identifier (e.g., `foo`)
- a literal (e.g., `42`, `3.14159`, `'a'`, or `"hello"`).

Non-tokens (such as white spaces or comments) are discarded. Invalid tokens (e.g., unterminated string literals) must be reported.

A *parser* takes a sequence of tokens as input and checks to make sure that they conform to the language specification. In order to pass this check, the input must have all matching braces,

semicolons, etc. Types, variable names and function names are not verified. A parser outputs a *syntax tree*, a tree representation of the program.

The Decaf Spec has the grammar of the language, which you will need to separate into a scanner specification and a parser specification. You will then implement the scanner and the parser, either by hand or using a scanner generator (e.g., `lex` or `flex`, or `jflex`) and a parser generator (e.g., `yacc`, `bison`, or `cup`) of your choice.

1.2 Semantic Checker

A *semantic checker* takes a syntax tree as input and checks to make sure that the program is well-formed, performing checks that cannot be done by the scanner and parser. For example, this checks that variables are declared before they are used, that variables have the right types, and that functions are called with the right number and types of arguments. The semantic checker will also build a symbol table in which the type and location of each identifier is kept. We'll supply a complete list of the checks in the project handout.

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.

1.3 Code Generation

In this assignment you will create a working compiler by generating unoptimized x86-64 assembly code from the IR 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 on a testing machine (more on the testing machines soon).

1.4 Data Flow Analysis

This assignment phase consists of building a data-flow framework to help optimize the code generated by your compiler. For this phase, you are required to implement the data-flow framework and a single data-flow optimization pass to test the framework. This framework will be used in the Optimizer project to build data-flow optimization passes.

We will provide a description of the framework and the required optimization be implemented in a later handout.

1.5 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 a 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 these programs, to identify which optimizations will have the highest performance impact. Your write-up needs to clearly describe the process you went through to identify the optimizations you implemented and justify them.

This phase requires a Project Design Document and a Project Checkpoint. The group has to provide two parts. First, a design document describing your design. This will be reviewed by the TAs and feedback will be provided in group meetings. This document will also count towards the project grade.

In this phase, the group has to submit a checkpoint of the implementation midway through the allotted time. The checkpoint exists to strongly encourage you to start working on the project early. If you get your project working at the end, the checkpoint will have little effect. However, if your group is unable to complete the project, the checkpoint submission has a critical role in your grade. If we determine that your group did not do a substantial amount of work before the checkpoint, you will be severely penalized.

1.6 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 one day before the Derby. This is done in order for your group to debug the compiler and get it working on this program. However, you are forbidden from adding any application-specific hacks to make this specific program run faster.

2 Grading

Make sure you understand this section so you won't be penalized for trivial oversights. The entire project is worth 70% of your 6.035 grade. You will turn in five times for the five segments and they are graded twice. The grade is divided between the segments in the following breakdown:

Scanner-Parser	Ungraded
Semantic Checker and Code Generator	25%
Data-flow Analyzer and Optimizer	45%

The remaining 30% comes from two quizzes, each worth 12%, and mini-quizzes at the beginning of every lecture, worth 6% in total.

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. Some parts of the project require additional documentation. Always read the *What to Hand In* section. Overall, a few pages for the supporting documentation is fine.
- (80%) Implementation. 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. Overall, we will limit the length of the supporting documentation to 8 pages (single-column, 11 pt font).
- (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.

3 What To Hand In

For each phase, you are required to submit your project write-up and complete sources (including all files needed to build your project). Your projects will be submitted via Github. Do not include compiled files. Instead, you repo should contain an executable file called build.sh in the top-level directory which, when run on an Athena machine with the appropriate lockers attached, will compile your code. These files are provided for you in the skeleton code; you may modify them if you need to.

Projects 2 through 5 will be done in groups. Each group will be given access to a repository for their project on Github.

There are few restrictions on how the project should be structured, except that it should be self-contained (apart from the allowed libraries and programming environment), and contain executables `build.sh` and `run.sh` in the root directory.

Option	Description
`-t|--target <stage>`	`<stage>` is one of scan, parse, inter, or assembly. Compilation should proceed through the given stage.
`-o|--output <outname>`	Write output to `<outname>`
`-O|--opt [optimization,...]`	Perform the (comma-separated) listed optimizations. `all` stands for all supported optimizations. `-<optimization>` removes optimizations from the list.
`-d|--debug`	Print debugging information. If this option is not given, there should be no output to the screen on successful compilation.

Table 1: Compiler Command-line Arguments

3.1 Command-line Interface

Your compiler should have the following command line interface.

```
./run.sh [options] filename
```

The command line arguments you must implement are listed in Table 1. Exactly one filename should be provided, which will not begin with a dash. The filename must not be listed after the `-O/--opt` flag, since it will be assumed to be an optimization.

The default behavior is to compile as far as the current assignment of the project and print the output to standard output unless different output is specified with `-o/--output`. All error messages should be printed to standard error.

By default, no optimizations are performed. The list of optimization names will be provided in the optimization assignments.

For each suggested language, we have provided code which is sufficient to implement this interface. It also returns a list of arguments it did not understand which can be used to add features. The TAs will not use any extra features you add for grading. However, you can tell us which, if any, to use for the compiler derby. You may wish to provide a flag which turns on only the optimizations you like.

3.2 Documentation / Write-up

Documentation should included in your source archive in the `doc/` folder. You will also submit one design document/report written for each of the five projects. Documentation should be clear, concise and readable. Acceptable formats include PDFs and plain text files.

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:

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.

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.

Difficulties. A list of known problems with your project, and as much as you know about the cause.

1. If your project fails a provided test case, but you are unable to fix the problem, describe your understanding of the problem.
2. 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.
3. If you were failing any test cases for a previous phase and managed to fix them by the current phase, give a short description of what the problem was and how you fixed it.
4. Describe any section of your project that you would like to highlight for more feedback on/had questions on.

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. (Projects 2 through 5 only.)