Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Fall 2014 Handout — Project Overview Wednesday, Sep 3

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, in Java or another allowed language, 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!

Important Project Dates (tentative)

Project Name	**Assigned/Due**	**Day**
Scanner and Parser	assigned: project public: project due:	Monday, Sep 4 Wednesday, Sep 17, 11:59 pm Monday, Sep 22, 11:59 pm
Semantic Checker	assigned: project due:	Monday, Sep 22 Thursday, Oct 2 11:59 pm
Code Generator	assigned: project due:	Thursday, Oct 2 Thursday, Oct 19, 11:59 pm
Data-flow Analysis	assigned: project due:	Tuesday, Oct 21 Monday, Nov 3, 11:59 pm
Optimizer	assigned: checkpoint due: project due:	Monday, Nov 3 Wednesday, Nov 26, 11:59 pm Monday, Dec 8
Compiler Derby	held on:	Tuesday, Dec 9

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

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 will use a tool called ANTLR (for JVM-based languages) to generate a scanner and a parser. The generated code will automatically perform most error checking and reporting for you.

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 on a testing machine (more on the testing machines soon).

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 to be implemented 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 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.

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.

Grading

Make sure you understand this section so you won't be penalized for trivial oversights. The entire project is worth 60% of your 6.035 grade. The grade is divided between the segments in the following breakdown:

Scanner-Parser	*ungraded*
Semantic Checker	9%
Code Generator	12%
Data-flow Analyzer	9%
Optimizer	30%

The remaining 40% comes from three quizzes, each worth 8%, 20 mini-quizzes at the beginning of every lecture, each worth 0.5%, and a 6% class participation grade.

The phases 2 to 4 of the project (Semantic Checking, Code Generation, and Data-flow Analysis) 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. We will limit the length of the documentation to 8 pages (single-column, 11 pt font).
- (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. Based on the testing, we will assign scores as follows:
  - Public Tests: 33%
  - Hidden Tests: 67%

The Optimizer project phase 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).
- (40%) 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 (10%). The hidden tests will check for overtly optimistic optimizations and incorrect handlqing of programs (30%).
- (40%) 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.

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). These sources should be places in a *.tar.gz* archive. These archives should be submitted on the course website.

This archive should not include compiled files. Instead, it 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.

`-t|--target= <stage>`	`<stage>` is one of scan, parse, inter, or assembly. Compilation should proceed to the given stage.
`-o|--output= <outname>`	Write output to `<outname>`
`-O|--opt= [optimizations]`	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

Command-line Interface

We will run your compiler with 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, and it should not begin with a '-'. 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 allowed 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.

Documentation / Write-up

Documentation should be included in your source archive in the `doc/` folder. It should be clear, concise and readable. Fancy formatting is not necessary; plain text is perfectly acceptable. You are welcome to do something more extravagant, but it will not help your grade. Acceptable file formats are pdf and plaintext.

Your documentation must include the following parts:

1. 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.)

2. A list of any clarifications, assumptions, or additions to the problem assigned. 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 TAs.

3. 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.

4. A brief description of interesting implementation issues. This should include any non-trivial algorithms, techniques, and data structures. It should also include any insights you discovered during this phase of the project.

5. 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. If this causes your project to fail hidden test cases, you may still be able to receive some credit for considering the problem. If this problem is not revealed by the hidden test cases, then you will not be penalized for it. It is to your advantage to describe any known problems with your project; of course, it is even better to fix them.

It is entirely up to you to determine how to test your project. The thoroughness of your testing will be reflected in your performance on the hidden test cases.