Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Spring 2019 — Handout — Project Overview — Tuesday, Feb 7

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) and second project (Interpreter) will be done individually. For subsequent projects, the class will be partitioned into groups of three students. You will be allowed to choose your own partners as much as possible. Each group will write, in C++, a compiler for a simple dynamically-typed programming language. We expect all groups to complete all phases successfully. Each subsequent project will build on your work from prior projects so do not fall behind!

Important Project Dates (tentative)

Project Name	**Assigned/Due**	**Day**
Scanner and Parser	assigned:	Feb 8
	project due:	Feb 19, 11:59 pm
Interpreter	assigned:	Feb 19
	Checkout due:	Mar 8, 11:59 pm
Low-Level Virtual Machine	assigned:	Mar 8
	checkpoints due:	Mar 22, Apr 5, 11:59 pm
	project due:	Apr 12, 11:59 pm
Garbage Collection	assigned:	TBD
	project due:	Apr 12, 11:59 pm
Code Generation	assigned:	Apr 12
	checkpoint due:	Apr 29, May 6, May 13, 11:59 pm
	project due:	May 15
Derby	held on:	May 16

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 an MITScript source file as an input and scans it looking for *tokens*. A *token* can be an operator (ex: "*" or "["), a keyword (`if` or `while`), 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. 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 may need to make some changes. You will use flex and bison to generate a scanner and a parser.

Interpreter

For this assignment, you will build an interperter for MITScript that given an MITScript program and its inputs, executes the program to produce the program's expected results. This assignment will include correctly understanding and implementing MITScript's semantics – such as the order in which arguments to a function are evaluated.

While your scanned and parsed program is syntactically correct, the program may still have a number of non-context free *semantic* errors. For example, the expression 5 / "hello" is syntatically correct, but does not have an obvious semantic meaning. Therefore your interperter will not only execute valid MITScript programs, it will also check and report errors for programs that have semantic errors.

This project will familiarze yourself with the full semantics of the MITScript language. Is it critical that you pay special attention to this project and deadline because successfully completing this project will give you a thorough understanding of the semantics of the language and will therefore help you focus more of your effort in the latter stages of the project on engineering a fast compiler versus understanding how to correctly execute the language.

Virtual Machine

In this project, your group will implement 1) a compiler from MITScript to a *bytecode* representation and 2) an interpreter for the bytecode representation – namely, a virtual machine. The virtual machine provides a low-level and language-independent abstraction for computation. The virtual machine accepts programs that consist of a list of functions with one such function designated as an entry. Each function consists of a list of bytecode instructions that manipulate and compute values using an *operand stack*. For example, the MITScript statement `x = 2 + y` translates to the following sequence of MITScript bytecode instructions:

```
int 2         # push 2 onto the operand stack
load_local 0  # load the value of y from memory and place it on the operand stack
add           # pop two integers from the stack and push their sum onto the stack
store_local 1 # pop the top value off the stack and store in x
```

In this project you will learn how to translate a high-level language down to a low-level, machine interpretable representation as well as understand how the design and performance of the abstractions in the low-level representation interact with and influence the design of the high-level language.

Garbage Collection

In this project, your group will implement a garbage collector for your virtual machine. As until now, we have not prescribed how your MITScript interpreter or your bytecode interpreter should handle the allocation of memory that corresponds to the data strutures in an MITScript program. It is not unreasonable to expect that your interpreter either consumes or leaks substantial amounts of memory.

In this project, you will address that problem by implementing a garbage collector. The garbage collector consists of a set of allocation routines that you will use to allocate runtime objects as well as a set of collection routines that will periodically scan the heap of your MITScript program and identify dead memory that can be freed.

If you find additional time during this of phase the project, take it as an opportunity to perform maintenance on your bytecode compiler and virtual machine to help make the next project phase more manageable.

Code Generation and Optimization

In this project, your group will generate x86-64 assembly code the bytecode of an MITScript program. We have structured this project such into two phases 1) a milestone at which you should demonstrate working – but perhaps unoptimized – code generation and 2) a final version of the project in which you demonstrate optimized code generation. For your milestone, you will also provide a write-up of the set of optimizations you plan to implement and the implementation strategy you plan to take in the second phase.

The second optimized code generation phase 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. Fo example, you can perform *data-flow optimizations* – such as constant propagation, common sub-expression elimination, copy propagation – *runtime representation optimizations* – such as more efficent record implementations or unboxing of primitive types – *efficient code generation techniques*– such as register allocation and peephole optimization – or even *just-in-time compilation techniques* – such as runtime specialization.

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.

In this phase, the group has to submit a couple of checkpoints of the implementation leading up to the project deadline. The checkpoint exists to strongly encourage you to start working on the project early. More details on how these checkpoints will work will be released with the project 5 description.

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(s) 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

The entire project is worth 75% of your 6.s081 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	0%
Interpreter	20%
Virtual Machine	15%
Memory Management	10%
Code Generation and Optimization	30%

Phases 2 to 4 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. Overall, a few pages for the supporting documentation is fine. An assignment may include written questions that we expect answered in your documentation.
- (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 TA first. Based on the testing, we will assign scores as follows:
  - Public Tests: 33%
  - Hidden Tests: 67%

Phase 5 of the project (Code Generation and Optimization) will be graded differently:

- (20%) Documentation, with particular attention given to your description of the optimization selection process, and examples of each optimization implemented.
- (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 hidden tests will check for overtly optimistic optimizations and incorrect handling of programs (50%).
- (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 (excluding the written questions) unless a problem arises, in which case you should contact your TAs as soon as possible.

Documentation

Documentation should be included in your source archive in the `doc/` folder, and you will have one design document/report written for each of the five projects. It should be clear, concise and readable. Fancy formatting is not necessary, just give us a clear idea what you did for each project. 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. This is the section where your optimizations will be explained as mentioned earlier in the handout. 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. 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. 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. (Projects 3 through 5 only.)