Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

This is an overview of the course project and how we will grade it. You should not expect to understand all of the technical terms, since we have not yet covered them in class. We are handing it out today to give you some idea of the kind of project we are assigning, and to let you know when the various due dates are. *Handout 6 - Decaf Language Definition* and *Handout 7 – Espresso Language Definition* describes the technical details of the project.

The class will be partitioned into groups of three or four people. You will be allowed to choose your own partners as much as possible. Each group is to write, in Java, a compiler for a simple programming language. Each group has a choice between a simple and a complex language; however, the logistics of the project are the same for each. We expect all groups to compete all phases successfully. In the event that your group does not fully complete an earlier phase (scanner or parser), you may use a staff-supplied implementation instead to build the later phases of your compiler.

♦ Important Project Dates

Wednesday, September 6	Scanner assigned. This is an individual project.
Monday, September 11	Scanner Project due Parser assigned
Tuesday, September 26	Parser Project due Semantic Checker assigned
Monday, October 2	Semantic Checker Homework due
Wednesday, October 11	Semantic Checker Project due Code Generation assigned
Thursday, October 26	Code Generation Checkpoint due
Wednesday, November 1	Code Generation Project due Data-flow Optimizations assigned
Monday, November 13	Data-flow Optimizations Homework due
Thursday, November 16	Data-flow Optimizations Checkpoint due
Wednesday, November 22	Data-flow Optimizations Project due Instruction Optimizations assigned
Monday, December 11	Instruction Optimizations Project due

♦ The Project Segments

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

1. Scanner

This part scans the input stream (the program), and encodes it in a form suitable for the remainder of the compiler. You will need to decide exactly what you want the set of tokens to be, and create the regular expressions for the scanner generator. The convention for this partitioning is quite standard in practice.

We will supply a scanner generator. This will consist of a program that takes a specification of the set of token types and outputs a Java program, the scanner. This specification uses regular definitions to describe which lexical tokens, i.e., character sequences, are mapped to which token types. The resulting scanner processes the input source code by interpreting the DFA (Deterministic Finite Automaton) that corresponds to the regular definition.

This is an individual assignment. Your scanner should work with both Decaf and Espresso languages.

The scanner is due on Monday, September 11.

When the group project starts, your group will need to choose a scanner to use. The group can use one of the scanners written by a member of your team. If you are not happy with any of the scanners written by your group members, you can use a scanner that we will provide for the later parts of your project.

2. Parser

This is the first group assignment. The parser checks the syntactic correctness of the token stream generated by the scanner, and it creates an intermediate representation of the program that is used in the code generation. You will also need to build the symbol table, since you will not be able to build the code generator without it.

We will supply a parser generator. This will consist of a primitive table-driven parser and a program that converts an LALR(1) grammar into a parse table suitable for use by that parser. You will have to transform the reference grammar into a LALR(1) grammar. We will also supply a Java based framework for the intermediate format representation. You are free to modify the IF representation to suite your needs.

The parser is due on Tuesday, September 26.

Similar to the scanner, if you are not happy with your grammar after you finish this part of the project, we will provide a grammar that you can use for the later parts of your project. However, your group must design and implement a symbol table on its own.

3. Semantic Checker

This part checks that various non-context free constraints, e.g., type compatibility, are observed. We will supply you with 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 will not 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.

The static semantic checker is due on Wednesday, October 11.

4. Code Generation

THIS IS A VERY TIME-CONSUMING PROJECT. For example, last year, it was done in two parts. YOU NEED TO START EARLY!

In this assignment you will create a working compiler by generating unoptimized MIPS assembly code from the intermediate format you generated in the previous assignment. First, the rich semantics of the Decaf and Espresso programs are broken-down into a simple intermediate representation. For example, constructs such as loops and conditionals are expanded to code segments with simple goto or jmp instructions. Next, the intermediate representation is matched with the Application Binary Interface, i.e. the calling convention and register usage. Then, the corresponding MIPS machine code is generated. Finally, the code, data structures and storage are laid-out in the assembly format. We will provide you with a description of the object language as well as a Java interface. The object code created using this interface can either be simulated using the SPIM simulator or assembled and executed on a native MIPS machine (e.g. athena SGIs).

This assignment has a checkpoint. At the checkpoint date, the group has to submit the code they have written so far. The checkpoint is there to strongly encourage you to start working on the project early. If you get your project working at the end, the checkpoint will have no effect. However, if your group is unable to complete the project then 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, i.e. waited till the last minute, you will be severely penalized.

The code generation checkpoint is on Thursday, October 26.

The assignment is due on Wednesday, November 1.

5. Data-flow Optimizations

This assignment will focus on optimizing the code generated by your compiler. You will implement a few traditional data-flow optimization algorithms to improve the quality of the code generated. We will provide you with a description of the optimizations that should be implemented. The object code created after this phase can either be simulated using the SPIM simulator or assembled and executed on a native MIPS machine (e.g. Athena SGIs).

This assignment also has a checkpoint. The checkpoint is on Thursday, November 15.

This data-flow optimizations assignment is due on Wednesday, November 22.

6. Instruction Optimizations

Modern architectures provide many opportunities for getting good performance. In this assignment you will implement a register allocater and an instruction scheduler to take advantage of a modern microprocessors.

This assignment is due on Monday, December 11.

♦ Grading

Make sure you understand this section so that you will not be penalized for trivial oversights.

The entire project is worth 54% of your 6.035 grade (and will solely determine your grade for the additional 6 extra units if you wish to implement Espresso, the more complex programming language). This breaks down to the individual parts as follows:

		6.035	**6.907**
1	Scanner	6%	
2	Parser	6%	12%
3	Semantic Checker	8%	18%
4	Code Generator	14%	28%
5	Data-flow Optimizer	12%	24%
6	Instruction Optimizer	8%	18%
		54%	100%

Each project in 6.035 will be graded on a 40 point scale, divided as follows:

- 10 points – Design and Documentation (subjective). Your score will be based on the quality of your design, the clarity of your design documentation, and the incisiveness of your discussion on design alternatives and issues. For some parts of the project, we will require additional documentation. Always read the **What to Hand In** section.

- 30 points – 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 your TA first.
    - 10 points – Public test cases. Points will be divided among test cases that we will make available. The public test cases demonstrate basic functionality of this phase of the project: in no way will they constitute comprehensive testing.
    - 20 points – Hidden test cases. Points will be divided among test cases that we will not reveal until all of the projects have been handed in.

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 part of the project, you will have to provide us with two things:

- online sources
- hardcopy documentation

➢ Online Source

Please see the scanner (lexical analysis) project handout for online source hand-in instructions for that segment.
For the remaining (group) segments, you will need to place your final code in your group directory. Specific directions for file layout will be available when you receive the first group assignment.

Note that you will be required to provide a `Makefile` which can generate your executable when your TA types `make`. (See Handout 4 for details on using `make`.) Remove all of the object files and try `make` to be sure it will work for your TA. This directory should contain class files that can be run by the TAs for testing purposes.

Files in your submission directory should not be modified after the deadline for submitting the project (5:00 p.m. on the due date). Please consult additional project handouts for phase-specific requirements.

➢ Command-line Interface

Your compiler should have the following command line interface.

```
java Compiler [ option | filename...]
```

where options are:

`-o <outname>`	write the output to <outname>
`-target <stage>`	<stage> is one of “scan”, “parse”, “inter”, or “assembly”; compilation should proceed to the given stage.
`-opt [ optimization...]`	preform the listed optimizations. "all" stands for all supported optimizations; "-<optimization>" removes optimizations from the list.
`-debug`	print debugging information. If this option is not given, there should be no output to the screen on successful compilation.

Exactly one filename should be provided, and it should not begin with a '-'. The filename must not be listed after the -opt flag, since it will then be assumed to be an optimization.

The default behavior is to compile as far as the current assignment of the project and produce a file with the extension changed based on either the target or ".out" if the target is left unspecified.

By default, no optimizations are performed.

The lists of names of optimizations will be provided in the optimization assignments; any which are requested but not supported will be mentioned.

You are provided with a class to handle this interface, CLI, which is sufficient to implement this interface and parse the command line. It also returns a Vector 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, but you can tell us which, if any, to use for the compiler derby. You may want to implement a "-O" mode, which turns on the optimizations you like.

For each assignment, your compiler should add support for the target of that assignment. The assignments will specify what should go in files output by each stage.

➢ Documentation

Hardcopy documentation should be turned in to the course secretary or your TA by 5:00 p.m. on the due date. 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.

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.

2. A list of any clarifications, assumptions, or additions to the problem assigned. The project specifications are fairly broad: they leave many decisions up to you, which is an aspect of real software engineering. If you think major clarifications are necessary, please consult your TA.

3. An overview of your design.
   - A description of your top-level design, 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.
   - A description on the design for **non-trivial** modules.
   - Any changes that 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. **Only for scanner and parser:** A listing of **commented** source code relevant to this part of the project. For later stages, the TAs can print out source code if needed. Do not resubmit source code from other parts of the project, even if minor changes have been made.

6. 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.

Recent graduates of 6.170 might be disappointed to know we don't require any documentation of evidence of your test cases or testing strategy. 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.