Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Fall 2005 　　　　　 Handout 3 — Project Overview 　　　　 Wednesday, September 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. Handout 5 describes the technical details of the project.

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 is to write, in Java, 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

Wednesday, September 7	Scanner-Parser Project assigned
Monday, September 19	Scanner-Parser Project due Semantic Checker assigned
Monday, September 26	Semantic Checker due Code Generation assigned
Tuesday, October 11	Code Generation due Data Optimization assigned
Monday, November 7	Data Optimization due Instruction Optimization assigned
Monday, December 5	Instruction Optimization due
Wednesday, December 7	Compiler Derby

The Project Segments

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

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

1

the regular expressions for the scanner generator. The convention for this partitioning is quite standard in practice.

We'll supply a scanner generator (JLex). 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.

Parser

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

We'll supply a parser generator (Java CUP). 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 an LALR(1) grammar.

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

THIS IS A VERY TIME-CONSUMING PROJECT. For example, in previous years, it has been done in two parts. YOU NEED TO START EARLY!

In this assignment you will create a working compiler by generating unoptimized x86 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 codegen 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 goto or jump instructions. Next, the intermediate representation is matched with the Application Binary Interface, i.e. the calling convention and register usage. Then, the corresponding x86 machine code is generated. Finally, the code, data structures, and storage are laid-out in the assembly format. We'll provide a description of the object language as well as a Java interface. The object code created using this interface will then be run on a testing machine (more on the testing machines soon).

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, 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.

The code generation checkpoint is scheduled on the calendar linked on the website.

Data-flow Optimizations

This assignment consists of optimizing the code generated by your compiler. We'll provide a description of the optimizations that must be implemented in a later handout. You may implement additional dataflow optimizations if you wish. As before, the object code created after this phase will be simulated using the SPIM simulator.

This assignment also has a checkpoint. Check the schedule!

Instruction Optimizations

In this assignment, you'll implement a set of instruction-level optimizations such as register allocation and instruction scheduling. These low-level optimizations are crucial for obtaining high performance on modern microprocessors.

This assignment also has a checkpoint. Check the schedule!

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. The remaining 30% comes from three quizzes, each worth 10%. The projects will be graded as follows:

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

- (90%) 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.

    - Public Tests (30%)
    - Hidden Tests (60%)

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.

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

Table 1: Compiler Command-line Arguments

What To Hand In

For each part of the project, you will have to provide us with two things: online sources and hardcopy documentation.

Online Source

Create a new directory with the project name (e.g., scanner-parser) at the top-level of your group locker. Copy all relevant source files to this directory. You are required to provide a Makefile that compiles your sources and packages the resulting *class* files into a java archive (see Handout 4 for details on using make). The *jar* file will used by the TAs to run the various tests.

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.

```
javac Compiler [option | 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 -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 produce a file with the extension changed based on either the target or ".out" if the target is unspecified.

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

We have provided a class, CLI, which is sufficient to implement this interface. 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. However, you can tell us which, if any, to use for the compiler derby. You may wish to provide a "-O" flag, which turns on the optimizations you like.

Hardcopy Documentation

Documentation should be turned in to the course secretary 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 and leave many of the decisions to you. This is an aspect of real software engineering. If you think major clarifications are necessary, consult your TA.

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. 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.

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