

CS544: Class1 Introduction

Introduction to the course

CS544 is a graduate-level compiler construction course. It covers both the theory and practice of compiler construction. You will be challenged to apply the theoretical and practical knowledge you have gained in other computer science courses and bring it to bear on solutions to the many difficult areas of compiler construction.

When you complete the course:

- You will have constructed a complete compiler for a non-trivial language.
- You will understand the structure of a language processor and the engineering decisions that must be addressed as one is designed.
- You will know many of the more important algorithms and data structures, and understand their benefits and disadvantages in compiler construction.
- You will be familiar with some of the compiler and programming language processing literature.

Secondary benefits of the course are:

- You will improve your proficiency with Java and various tools, such as Eclipse and SourceForge.
- You will improve your general problem solving ability.
- You will increase your confidence in working with significant software systems.

Administrivia

There are several things you need to know about the course, including my expectations for you, the amount of work, and how you will be evaluated. The following sections describe each of these.

Grading

Grades are a necessary consequence of the academic environment. While it would be wonderful if everyone were able to receive an A for the course, that probably will not happen. I do not grade on a curve. What you earn is what you get. Therefore, you should prepare yourself as best as you possibly can for each area of the grade. The grades will be determined approximately on the following criteria. I do reserve the right to change the criteria as appropriate.

- Homework - 25%
- Quizzes - 15%
- Exams - 25% (combined, mid-term and final)
- Project - 25%
- Class participation - 10% (e.g., leading discussion or making a presentation on the readings and ability to answer questions)

Expectations

There are several expectations I have of the student who successfully completes the course.

- You should expect to spend approximately 15 hours per week on this course.

- I expect you to keep up with the readings. There will be readings from the textbook and additional papers assigned.
- I expect you to be prepared to come into the class able to participate in class discussions.
- I expect you to do your own work. If you are working on a team, I expect you to contribute to the team as much as your teammates do.
- I expect you to ask for help and use my office hours and other times and means of contacting me when you need it. It is your responsibility to seek out help when you need it. I am fully ready to help, but I need to know when you need help.

Office hours

Office hours are posted on my home page (<http://www.cs.wpi.edu/~gpollice>) and outside of my office.

Contact

There are several ways to contact me.

- At my office. I have regular office hours when you can expect to find me in the office. However, if you need to see me, come by. My door is usually open and I'm able to talk. The worst that can happen is that I would tell you to come back later.
- e-mail (gpollice@cs.wpi.edu)
- AIM ([gpollice](#))
- MSN Messenger (gpollice@rcn.com)
- Phone (508-831-6793)

Tools needed

Eclipse

See document: www.eclipse.org

Your work will be done using Eclipse as your development environment. If you choose not to use Eclipse, you are still responsible for turning in your programming work as an Eclipse project. Therefore, you will need to port the work from your preferred environment to Eclipse.

Setting up Eclipse

If you are new to using Eclipse, or need a refresher on how to install it and create projects, you can view the [Eclipse setup presentation](#) and the presentation on how to [create a project in Eclipse](#).

SourceForge

See document: sourceforge.wpi.edu

Using SourceForge

See document: [CreateSourceForgeProject.htm](#)

In order to use SourceForge, you need to have an account on SourceForge. If you don't have one, you can get one by going to <http://sourceforge.wpi.edu>.

SourceForge has adequate on-line help. If you want to have a quick introduction to creating a project and setting up a CVS repository, look at the presentation on [setting up your SourceForge project](#).

The master project

I have created a master project for this course. All of the materials you need, such as source code and additional readings, etc. will be in this project.

Each class member will be given read-only access to this project.

The project is called **CS544 Course Materials**.

JavaCC

See document: javacc.dev.java.net

JavaCC is a parser and scanner generator built in Java. It generates a top-down, recursive descent parser and quite a flexible lexical analyzer. This is the main compiler generation tool that we will use for the course.

We will examine bottom-up parser generators like yacc, and scanner generators like lex, but JavaCC will be our main tool. It is reasonably easy to use and provides a lot of freedom for your design. It also comes with a tree builder and documentation tool.

JavaCC is an open source software tool. The home page for JavaCC is <https://javacc.dev.java.net/>. There is a presentation you can view that shows you how to obtain and set up JavaCC, along with the [Eclipse plug-in](#) we will use to work with JavaCC.

Hardware simulators

A significant part of the course will be spent on optimization and code generation. In order to enable you to concentrate more on the underlying principles, we will use the MIPS simulator, SPIM to test the code generated by our compilers. More information about SPIM will be presented as necessary.

Resources

Textbook

See document: [subindex.asp](#)

The only text you need for this class is [Engineering a Compiler](#), by Cooper & Torczon. This text contains up-to-date information on compiler construction. It has an excellent coverage of code generation and optimization topics. These topics are often omitted or given only cursory treatment in many textbooks. Yet, these are perhaps the most important topics in modern compiler construction.

Other texts

You do not need any other textbook for the course. However, there are several good ones that I can recommend if you want to have an alternative discussion to some of the topics, or need to see special coverage of a specific topic.

Some of the assignments may be taken from these books, but if they are, you will be given everything you need to complete the assignment.

You will note that I haven't included the Aho, Sethi, and Ullman "Dragon Book" that has become a classic. While it is still an excellent reference for the theory of parsing, it does not address many of the issues of modern compiler construction. Any professional compiler developer should have it in his or her library, but it is not one of my top choices for a text for this course.

Modern Compiler Design

See document: [0,1144,1576761053,00.html](#)

Modern Compiler Design by David Galles is a nice text on basic compiler design. It contains a good description of JavaCC and how to use it, and a nice discussion of some of code generation and memory management.

Modern Compiler Implementation

See document: [java](#)

Andrew Appel's book, [Modern Compiler Implementation in Java](#) (he has the same book for implementation in C++ and ML) is an excellent, all-around compiler text. It was my second choice for this course. He uses a different set of tools, which I consider a bit more difficult than the ones we are using. He does have a very good discussion of code generation, optimization, and memory management. His treatment of the front-end portions of the compiler.

One noteworthy set of topics are his chapters on object-oriented and functional language processors.

This book is on reserve for this class in the library.

Programming Language Processors in Java

See document: [0,1144,0130257869,00.html](#)

[Programming Language Processors in Java](#), by Watt and Brown, is a fairly easy treatment of language processors in general. It does have the advantage of a good [companion site](#) that has code and tools. It focuses on recursive descent parsing and code generation for a simple stack machine. The discussion of abstract syntax trees is worth reading.

This book is on reserve for this class in the library.

Modern Compiler Design

See document: [description.cfm](#)

Another book with this name, [Modern Compiler Design](#), by Grune et. al., is an interesting text. It has a somewhat terse, yet complete treatment of many topics, such as lexical analysis, both by hand and with a generator. If you are considering writing a language processor for functional, or logic programming languages, there are very good chapters on those topics.

This book is on reserve for this class in the library.

Readings

The outside readings for this course will be specified either by URLs pointing to the specific paper, or the paper will be available in the SourceForge project.

Class format

Lecture

There is a significant portion of class time that will be devoted to lecture. We will cover many topics in the course. Several of these -- at least one from each major topic area -- will be covered in depth. You are expected to read the text that corresponds to the lecture material before the class meeting.

Readings discussion

The assigned readings are an important part of the course. Each student will be expected to lead the discussion of at least one of the readings during the semester. This means that you are expected to come to each class where readings are discussed prepared with a set of discussion questions and opinions about the material.

Any discussion should include:

- What was the reading about?
- Why is it important?
- What difficulties are there?
- Do you agree with the reading
- Was it worth reading?

Quizzes

During the term there will be short (10 min.) quizzes that will give you an opportunity to show your progress in the material in the textbook and previous lectures. There will be a quiz approximately once a week.

Project

The project for this class is a major part of your grade. You may work in teams of two or three students if you wish, but are not required to, although I highly recommend it. If you work in a team, there will be a basic grade assigned to the project, and adjustments to that grade will be made based upon each team member's contributions.

The project is the construction of a complete compiler, or a significant extension to an existing compiler. This will be determined by the first class.

Compilers overview

There are many aspects to be considered in the study of compilers. Usually the study encompasses more than just the strict definition of a compiler. According to Webopedia, a compiler is:

A program that translates source code into object code.

The textbook, Cooper & Torczon (CT) says that a compiler "takes as input the specification for an executable program and produces as output the specification for another, equivalent executable program."

Language processors in general

We will concern ourselves with the study of the principles of compilation. However, the application of these principles and techniques are widely used in many other contexts. Examples of where you will find the such use are:

- static analysis tools
- testing tools
- XML tools for understanding and producing valid XML documents
- program rewriting and refactoring tools
- debuggers

For the purposes of this course, we will group all uses of the techniques into the category of *language processors* since we will be mostly concerned with understanding programming languages.

Relation to other parts of CS

The study of compilers brings together many of the theoretical and practical subjects you have studied in your computer science career and give you one example of how they work together for a very important, practical application.

Algorithms

Many algorithms have been developed specifically to enable more efficient compilers. Graph coloring algorithms for register allocation are one example of the type of algorithm you will encounter.

Software Engineering

A compiler is usually a fairly large program that is composed of several components. It must be planned, designed, implemented, tested, and delivered according to the best software engineering practices.

You will be expected to manage your project by producing appropriate requirements, tasks, and tests. You are also expected to use iterative development, with each iteration no longer than two weeks long. At the end of each iteration, you are expected to deliver some increment of working software.

Engineering a compiler involves tradeoffs that must be evaluated intelligently. For example, what are the implications of certain types of optimizations on the compilation time? Are they worth the effort? Typical tradeoffs in compiler design are:

- speed of compilation
- size of the generated code
- speed of the generated code

Foundations

Automata theory plays a crucial role in lexical analysis. All scanner generators are based upon the results of automata theory.

Properties of language, such as context-free languages are exploited in parsing theory.

Design

Compilers must be designed. They require explicit interfaces between the various components. Since we will be using Java, we should expect to use abstraction, encapsulation, and polymorphism along with appropriate design patterns.

Computer architecture

A compiler that produces some form of executable code must take into account the underlying machine architecture, even if that architecture is a virtual machine like the Java Virtual Machine (JVM).

Compilers are expected to generate efficient code. Code generating patterns used for one type of architecture (e.g., RISC) might be counterproductive and inappropriate for another type of architecture (e.g. CISC). The compiler writer must know how instructions are scheduled and processed in the target machine in order to produce an effective instruction sequence.

Correctness

Compilers are probably different than any program you have ever worked on. We strive for programs that are efficient and defect free. However, we have come to accept the fact that there will be defects creeping through our development process, into the actual release.

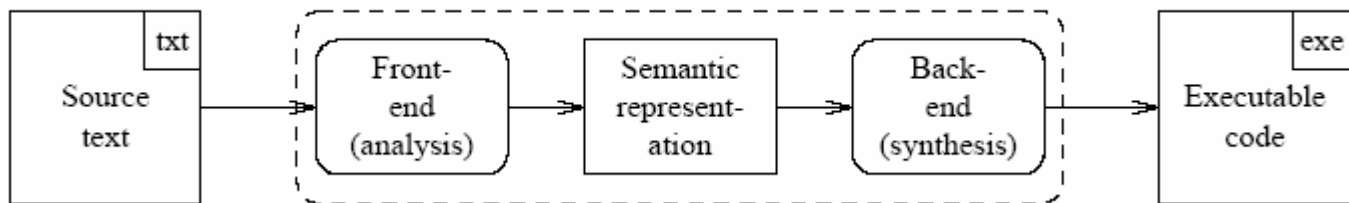
The demands put upon a compiler for correctness are about the strictest you will encounter, except possibly for software that is responsible for safety-critical applications. It would not do to have a compiler that compiles programs correctly 99.999% of the time. It must produce a compiled program that is *exactly* a semantic equivalent of the source program and that executes properly on the target hardware.

This correctness requirement is especially important when you are working on optimizing the program. Each optimization must produce a program that is semantically equivalent to the original program in all cases. For example, if you determine that it is acceptable to eliminate instructions to store a value to a memory location, you must ensure that this value is not going to be accessed via another thread. If there is any possibility that it is, you may not make the optimization.

Compiler writers must take the most conservative view possible of the techniques they use. One way of making sure that your program transformations are acceptable is to use the formal semantics of the programming language specification and develop a *mini-proof* of the transformation. If you can prove the validity of your transformation, then you are free to implement it.

Structure of a compiler

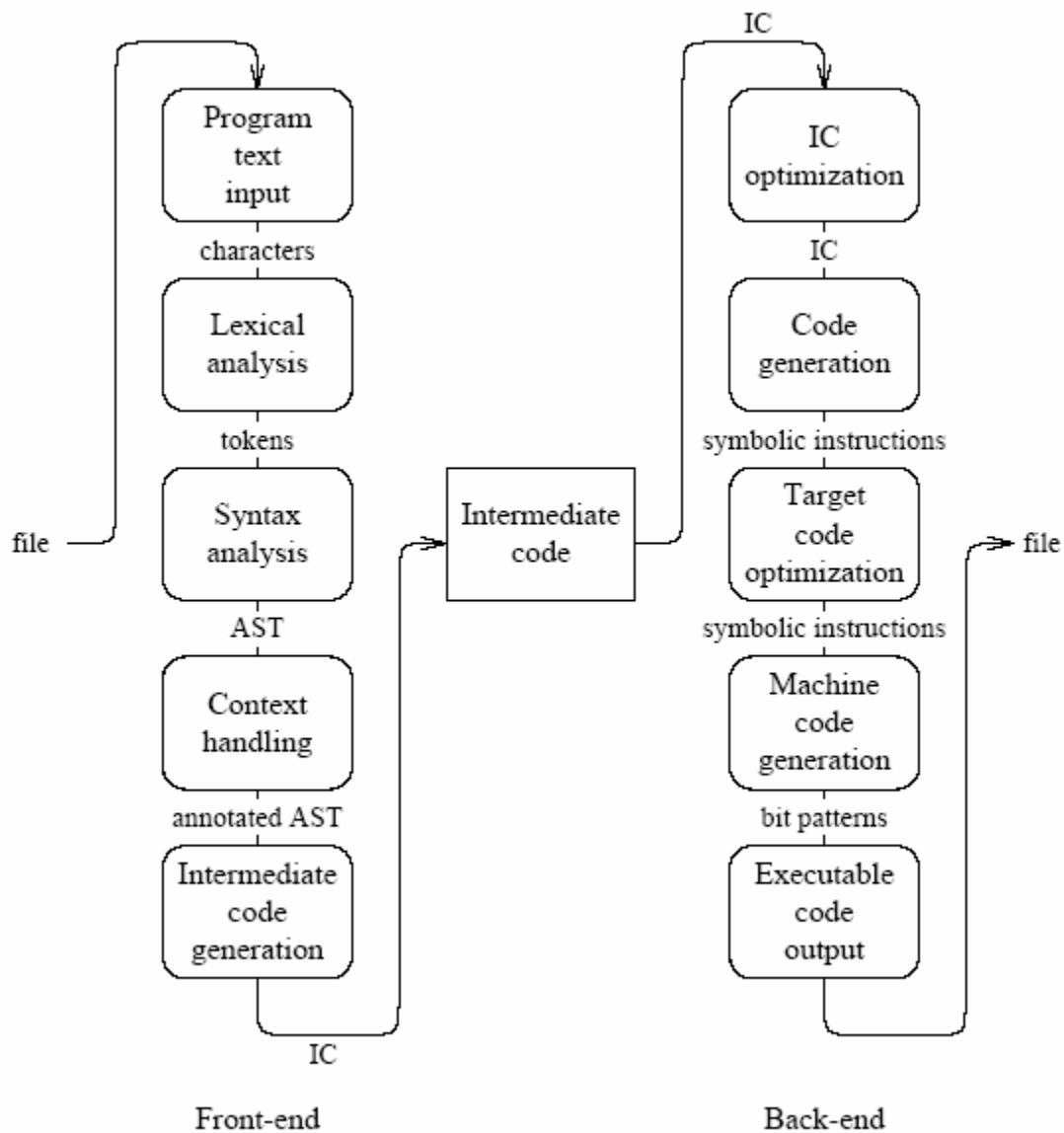
The general structure of a compiler is shown below (diagrams in this section are taken from Grune, et. al.



You start with one representation of the program (using a very broad interpretation of what we mean by "program"). That representation, the source, is analyzed for structural correctness by the lexical analyzer (scanner) and the parser. The parser cooperates with the semantic analyzer to ensure that the program is not only structurally correct, but meaningful in terms of the source language's semantics. These actions are typically referred to as the compiler's *front end*.

One of the challenges of compiler design is the communication mechanisms for each interface between the different parts of a compiler.

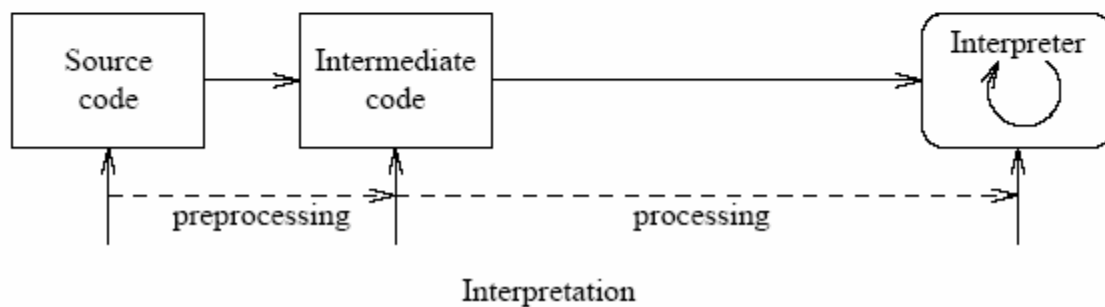
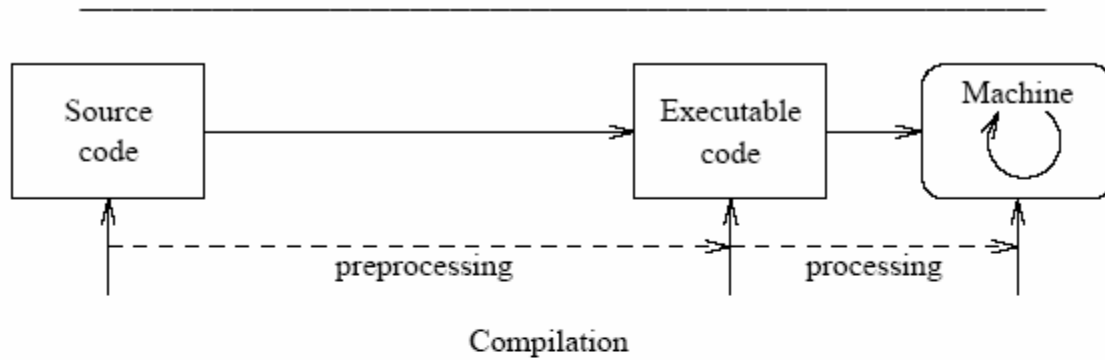
The following diagram is a more detailed look at the structure of a typical compiler.



Along with communication issues, you need to decide whether each module processes a complete compilation unit before passing it onto the next module, or if they all work in a cooperative manner on small parts of the input at any given time. This is called the compiler's *bandwidth*.

Compiler vs. Interpreter

An interpreter translates some form of source code into a target representation that it can immediately execute and evaluate. The structure of the interpreter is similar to that of a compiler, but the amount of time it takes to produce the executable representation will vary as will the amount of optimization. The following diagram shows one representation of the differences.



Compiler characteristics:

- spends a lot of time analyzing and processing the program
- the resulting executable is some form of machine- specific binary code
- the computer hardware interprets (executes) the resulting code
- program execution is fast

Interpreter characteristics:

- relatively little time is spent analyzing and processing the program
- the resulting code is some sort of intermediate code
- the resulting code is interpreted by another program
- program execution is relatively slow

The above characteristics are typical. There are well-known cases that are somewhere in between, such as Java with it's JVM.

Front End Structure

A typical structure for the front end of a compiler consists of three parts:

- The lexical analyzer, or scanner, that converts the source code into a stream of *tokens*. Each token represents the occurrence of a single lexical element, called a *lexeme*, in the source program.
- The parser that analyzes the token stream to ensure that it is a instance of a string that is in the source language.

- The context, or semantic, analyzer that ensures that the input is meaningful. For example, it ensures that all variables that are referenced have been declared (if that is a requirement of the language).

Middle Structure

Most modern compilers have an optimization phase that sits between the front end and back end of the compiler. Usually, the optimizer consists of several cooperating modules that take, as input, some intermediate form of the program under compilation and produces a transformation of it that is (hopefully) optimized.

The optimization phases of most compilers today are composed of several modules because it is much easier to design and maintain the optimizer than if it were implemented as a single, monolithic, module that tried to do all optimizations.

Back End Structure

The back end of the compiler is responsible for emitting the final (executable) version of the source program. Typical parts of the back end are responsible for:

- instruction selection
- register allocation
- memory management
- instruction scheduling

Common Infrastructure

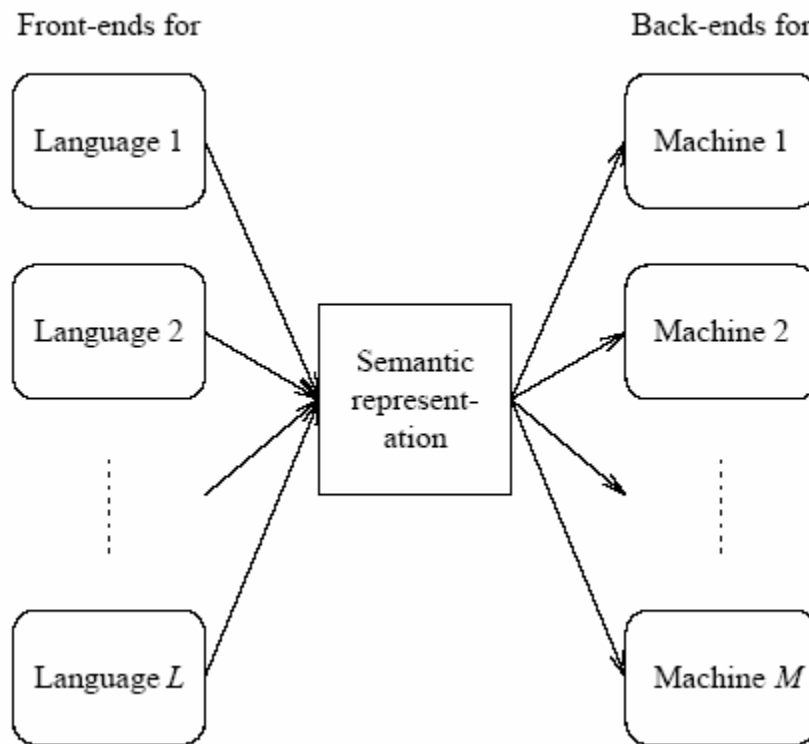
A compiler will have certain modules that are used to collaborate with modules in the front, middle, and back of the compiler. These typically consist of:

- symbol tables
- grammars
- trees
- graphs
- other data structures

Avoiding component proliferation

One of the problems that confronted many early compiler writers was how to minimize the number of components needed to support a set of programming languages on multiple computer architectures. For example, if you were going to have L languages that needed to run on M machines you would conceivably need L front ends and M back ends for each of the front ends. This means that you would need a total of $L + L * M$ total components.

The solution to this problem is to have a standardized intermediate representation that is produced by each front end and consumed by each back end. This then requires you to only have $L + M$ components, as shown in the following diagram:



A simple interpreter

Writing a language processor is more than hacking together a few lines of code in a scripting language that transforms one form of a program to another. Even simple processors require some thought as to how the different parts will be distributed and collaborate.

We will look at an example interpreter for a simple calculator and see how it is constructed, the tests, and how it evaluates expressions.

Calculator: hcalc1

The calculator we will look at is in the SourceForge CS544 Course Materials project in the `cs544.hcalc1` directory. "hcalc" stands for hand-written calculator, rather than one that uses JavaCC to generate a significant portion of the program.

The calculator is very simple. It only allows addition and subtraction and integer operands. It accepts expressions on a single line of text and when the end of line is reached, it evaluates the line and produces the result. There is also a simple error handling and reset capability.

We will do a code walk through for you to get a feel for the structure of the program. You will modify this program for your first assignment.

Assignment

Programming: Modify the calculator in `hcalc1` to add multiplication and division. For now, do not worry about precedence. That is, the expression: $2 + 2 * 3$ will evaluate to 12, not 8. It will be evaluated as $(2 + 2) * 3$.

Also modify the calculator to have floating point numbers. The numbers should be simple floating point numbers of the form: Integer . Integer. Do not worry about scientific notation.

The program is due as an Eclipse project, with tests. I should be able to add my own test file to the project, like the original CalcParserTest.

The assignment is due by the beginning of Class 3.

Reading

Read: Chapter 1. Pay special attention to the description of ILOC. You should be able to read ILOC sequences without problems. Chapter 2, sections 2.1-2.3.