Fifteen Compilers in Fifteen Days

Jeremy D. Frens
Calvin College
1740 Knollcrest Circle SE
Grand Rapids, MI 49546-4403
jdfrens@calvin.edu

Andrew Meneely
Calvin College
Zeta 7
Grand Rapids, MI 49546
apm2@calvin.edu

ABSTRACT

Traditional approaches to semester-long projects in compiler courses force students to implement the early stages of a compiler in depth; since many students fall behind, they have little opportunity to implement the back end. Consequently, students have a deep knowledge of early material and no knowledge of latter material. We propose an approach based on incremental development and test-driven development; this approach solves the emphasis problem, provides experience with useful tools, and allows for such a course to be taught in a three or four weeks.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education; Curriculum*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Test-Driven Development*

General Terms

Design, Human Factors, Languages

Keywords

compiler course, incremental development, refactoring, test-driven development, unit testing

1. INTRODUCTION

Calvin College has experienced declining computer-science enrollments over the past few years (like everyone else); while these enrollments are likely to increase as they have in the past, we anticipate a larger growth in our information-systems major. Consequently, our compiler course may never be offered again, especially when students prefer taking more practical (or less frightening) upper-level electives.

Calvin has an interim term (a.k.a. a “January term”) when students take one class for three hours a day for fifteen days. The courses are typically pass/fail, so it can be a time for faculty and students to explore different, new, or off-beat areas of their discipline. Since an interim course moves *so* quickly, it seems dangerous to take on a large software project, and it seems particularly ludicrous to take on a large software project that also requires significant learning or background knowledge—a compilers course is right out!

Or is it? Certainly the concepts behind a compiler could be taught, and the programming assignments adjusted. An Artificial Intelligence course could be taught during interim (as we do at Calvin). The focus of such a course is on conceptual learning from lectures and the textbook; students implement just a few, smallish programs—fewer and smaller than they might during a normal semester. Similarly, a compiler course could stress the concepts of computer design, and programming assignments could be isolated portions of a compiler. Many compiler instructors already take this approach with the programming assignments. However, Calvin’s tradition is to implement a complete compiler from front to back—a tradition not given up lightly by students or faculty.

A traditional approach to a front-to-back compiler has students implement the stages of the compiler in the same order in which they are used: a full lexer, then a full parser, then a static-analysis stage, and then code generation. Fitting this into an interim would be impractical: students will inevitably fall behind, and most students will not reach the latter stages of their compiler at all. Arguably, these latter stages are more interesting, making the loss more regrettable.

While contemplating our enrollment problems and the insanity of putting a compiler course into the interim, one of us (Frens) proposed using agile software development and an incremental approach to writing a compiler: first write a *complete* front-to-back compiler for a simple language; add a new feature to the language and rework the compiler; repeat. This approach seems workable as an interim course. Students see a complete compiler their first day and every day after. Even when students fall behind in the course, they have seen several complete compilers and have learned the basics of each stage of a compiler. Missing the latter features of the project means missing advanced topics, but this is acceptable since advanced topics are less important for the average student.

Since this was not an idea to take directly into an interim, Frens recruited Meneely, a student, to try this approach as an independent study during a full semester. This paper is a description of what we did and of what we think should be done. We offer this paper as a proof-of-concept and as a guide for others.

This paper is organized into five sections. The next section describes the tools used in this incremental approach. The third section describes our independent-study course, including suggestions for doing it better. The fourth section describes some of the benefits of this approach. The last section provides some concluding thoughts, including future projects.

2. OUR TOOLKIT

Incremental development builds a complete system at each stage of development. Only a few features may be added each time so that each stage is completed in a short period of time. No consideration is made for what *might* be implemented in future stages; only the *current* feature set matters.

We used several tools in our development which made it possible to take an incremental approach to developing a compiler in an interim course. Understanding these tools is critical to our claim.

2.1 Test-Driven Development

The key tool in incremental development is test-driven development (TDD) [2]. TDD follows this process:

1. Write a unit test for a new ability in a class.
2. Add enough code to get rid of compiler errors but *without* any useful computations.
3. Run the tests, and see the new test fail.
4. Repeat until all tests pass:
   (a) Refactor.
   (b) Write computation code to add the new ability.
5. Refactor.

Put enough new abilities together, and the compiler will have a new feature, front-to-back.

The two key components in this process, unit testing and refactoring, are described in the next two sections.

2.2 Unit Tests

The tests in TDD are unit tests [2, Part II]; we used JUnit [11][2, Chapter 4]. A unit-test tests one small bit of the code. For example, the lexer should recognize the `if` keyword:

```
assertToken(IF, "if", "if");
```

This assertion uses a special JUnit extension for lexers and parsers [1]. `IF` is a special token type created by the lexer; the first `"if"` is the expected text of the returned token; the last `"if"` is the text to scan.

Every small ability of the code is unit tested at this level; sometimes this means lots of unit tests for what seems like a simple feature. For example, when testing the code generation for an integer literal, it is not enough to test just one integer. A load-immediate instruction might be good enough for small integers; larger integers might require two or more instructions. It is also necessary to make sure that small negative integers and large negative integers work as well. This means at *least* four different unit tests.

The unit tests are run on a regular basis; generally whenever new code is introduced or code is changed. All of the tests are executed, serving as regression tests so that old abilities are not silently broken. Even if a bug does slip through the unit testing, a new unit test can be added to demonstrate the bug so that the bug is never happens again.

2.3 Refactoring

TDD works best (or at all) only when it includes refactoring [12]—changing the code without changing its effect. The purpose of refactoring is both to simplify the code and to make it more readable.

For example, consider the `assertToken()` method mentioned above. Originally, three statements were needed to fully test a token:

```
public void testIfKeyword() {
  Token token = makeLexer("if").nextToken();
  assertEquals(IF, token.getType());
  assertEquals("if", token.getText());
}
```

Three statements like these would be repeated for *each* keyword. So, before implementing any other keywords, the common code in the existing test is refactored into a new method for further reuse.

The first step is to generalize these three statements. Introduce Explaining Variable [12, pp. 124–127] is used three times to create three temporary variables:

```
public void testIfKeyword() {
  int type = IF;
  String text = "if";
  String input = "if";
  Token token = makeLexer(input).nextToken();
  assertEquals(type, token.getType());
  assertEquals(text, token.getText());
}
```

Extract Method [12, p. 110–116] is a very powerful refactoring to create a new method based on existing code. The last three lines of the test method can be extracted into a new method:

```
private void assertToken(int type, String result,
      String input) {
  Token token = makeLexer(input).nextToken();
  assertEquals(type, token.getType());
  assertEquals(result, token.getText());
}
```

The original method can use this new method:

```
public void testIfKeyword() {
  String type = IF;
  String text = "if";
  String input = "if";
  assertToken(type, text, input);
}
```

The test method can be simplified even more by inlining the local variables:

```
public void testIfKeyword() {
  assertToken(IF, "if", "if");
}
```

Now the new method can be reused for testing the lexing of other tokens. The work spent on this refactoring is amortized by the number of times it is used in the future.

Martin Fowler's refactoring book [12] (i.e., *the* refactoring book) has over seventy different refactorings, and it is certainly not an exhaustive list. Refactorings vary, of course, in their usefulness and frequency, but even a familiarity with a dozen refactorings is enough to be an effective test-driven developer.

2.4 Test-Driven Development Revisited

Refactoring relies heavily on the tests. Not all refactorings preserve all behaviors, and the unit tests serve as a safety net: if the tests fail after a refactoring, the refactoring should be undone, and a different refactoring should be tried.

Refactoring is necessary because the code from TDD is written for only the current set of features; there is no up-front design. When a new feature is added, the code needs to be refactored before adding the new feature. Extracting an interface and extracting methods (and other refactorings) generalize the code so that the feature can be added; renaming variables and methods are very helpful in putting the programmer in the right frame of mind to add a feature. Once a new feature is added, there may be new opportunities for refactoring like extracting a method to eliminate redundant code or renaming a variable to more accurately describe its responsibilities..

3. OUR COMPILER COURSE

We met twice a week for "lecture" which mostly consisted of discussing concepts from the textbooks [3, 13] and discussing our compilers. We also met a third time each week for a two hour "lab" in the department's computer lab; this "lab" proved to be particularly effective. Our goal was to develop a new compiler each week of a normal semester; during an interim, it would be a new compiler each day—hence our catchy "15 compilers in 15 days" mantra.

We implemented our own compilers in Java (J2SE 5.0) using the Eclipse IDE [5, 7] (version 3.0). We used ANTLR [14] (version 2.7.5) to implement a front end. As mentioned above, unit testing was done with JUnit [11][2, Chapter 4] (version 3.8.1) and a JUnit extension for testing ANTLR grammars [1] (alpha version).

Our target language was PowerPC assembly code, and this was facilitated greatly by IBM's *The PowerPC Compiler Writer's Guide* [9].

3.1 The First Three Compilers

A description of our first three compilers is instructive of our proposed approach.

Compiler #1. We started with a compiler for an integer language. That is, given a file with an integer in it, the front-to-back compiler would read this file and generate PowerPC code to print this integer.

Immediately with this first compiler, there are some judgment calls that needed to be made. Java's `readLine()` might work fine for this first compiler unless ignoring whitespace is part of the initial feature set. Since ignoring whitespace *was* part of our feature set, we opted to use a formal lexer from the beginning. A parser, on the other hand, was not necessary since the language so far had no deep structure.

So the front end of this first compiler consisted of a lexer which produced an `INTEGER` token. The back end turned an `INTEGER` token into a PowerPC program to print the integer value.

All together, this is actually a very significant amount of material to learn: simple regular expressions (enough to describe integers), PowerPC assembly code, lexer syntax, and the lexer's API.

Compiler #2. For a second compiler, we added string literals to our feature set. Now using a lexer really paid off, and it involved learning some new features of ANTLR (e.g., how to ignore the double quotes). Generating assembly code also became more interesting. Overall, this was very straightforward, and it might not have been ambitious enough!

Compiler #3. For the third compiler, we implemented infix arithmetic expressions.

Arithmetic expressions demand a parser. ANTLR's data structure for an abstract syntax tree has an awkward API, and we wanted to use the visitor pattern [8, pp. 331–344] to implement the algorithms for static analysis and code generation. Consequently, we also implemented an ANTLR tree-builder to turn the parser's input into our own "tree intermediate representation" (TIR). So our front ends consisted of three phases: a lexer, a parser, and a tree builder.

Since the output from the front end changed significantly (i.e., from lexer tokens to tree-builder TIRs), we spent a good deal of time refactoring the code for the back end. This change is actually instructive since the token-based back end was primarily procedural—a `switch` statement over the type of token. By switching to TIRs and the visitor pattern, we had *much* better object-oriented code.[1]

An interesting opportunity arises with this third compiler: constant folding. The language does not support variables yet, and if one wanted to avoid learning how to do arithmetic in PowerPC assembly code, one could learn about and implement optimization on the *third* week of class!

Considering all that this third compiler involved, perhaps it is not surprising that this took us more than one week. In hindsight, arithmetic expressions should have been done in stages—first binary expressions, then multi-term expressions, then associativity, then precedence, etc. Essentially, we *did* use these stages to implement the third compiler, but it would have helped to make these stages explicit in our planning.

While working on a constant folder and other parts of this third compiler, we discovered issues we had not thought of originally.[2] Omissions are to be expected and should be embraced—it is what actually happens to developers, and we learn more from our mistakes.

3.2 The Rest of the Compilers

Adding `let` expressions made symbol tables necessary and introduced the idea of scope. It also meant that the constant folder could not implement all of our arithmetic anymore. We should have spent two weeks on adding `let` expressions, breaking it down into smaller features.

Compound statements, `if` statements, and `while` statements all took about a week. Adding *three* control-flow statements sounds ambitious, but a `while` loop is just an `if` with some extra `goto`s.

[1]By the end of the project, because of the number of algorithms implemented over TIRs, students will appreciate the visitor pattern and OOP in general.

[2]Like handling negative integers, for shame!

We used destination-driven code generation [4]. Special approaches to compiler design like this have to be carefully introduced in the feature sets.

Meneely implemented `for` loops but got distracted by the PowerPC's count register. It turned out to be much easier to translate the `for` loop into a `while` loop as part of the compiler's static analysis.

Late in the semester, we tried to implement a linear intermediate representation (LIR), but that proved to be too ambitious, mostly because we did not have an existing library of LIR instructions.

3.3 Teaching the Concepts

In the end we covered most of the conceptual material in our "lectures" that would normally be covered in a compiler course. The same could certainly be done during interim where there are three hours of lecture per day as opposed to our two hours per week. However, this implies that students spend several hours *outside* of lecture working on that day's compiler.

One significant change to the lectures is the order in which the material is taught. The first day of interim, students need to know a *little* about regular expressions and lexers, but perhaps the focus needs to be on assembly code. Parsing could be skipped until needed the third (or even the fourth) day, and some static analysis could be covered earlier.

The more flexibility students are given to pick their own feature sets, the harder it is to keep the lectures synchronized with everyone's coding needs. The instructor also has to be more flexible to cover the material as needed or restrict the students' options.

3.4 Good Things That We Did

Using Eclipse as our IDE was one of the smartest things we did. The JUnit plugin is useful; an ANTLR plugin facilitates the editing and processing of ANTLR files. Eclipse also provides a lot of automatic refactorings: rename anything (method, variable, class, package, etc.), extract local variable, extract method, extract constant, inline variable, inline method, etc., etc., etc. Automatic refactoring tools make the easy refactorings *really* easy and make the hard refactorings possible.

We used CVS (Concurrent Versioning System) so that rolling back to earlier versions of the compiler was not a problem. (Eclipse has wonderful CVS support which also helped.) A versioning system is important when refactoring. A failed refactoring (i.e., it generates failed unit tests) may need to be undone by restoring an earlier version. Occasionally a sequence of refactorings actually make the code *worse*, and then a CVS backup is essential.

Using a tool like ANTLR for the front end was the right call. We found it relatively easy to refactor the grammar. Implementing the parser itself would be considerably more work than manipulating grammars.

One of us (Frens) had already written classes for TIRs and interfaces for visitors on the TIRs. To ask students to implement this code is busy work that detracts from learning about compilers; however, it violates the "build only what you need" mantra of agile development[3]. The trade-off seems worthwhile for pedagogical reasons.

[3]E.g., many of the methods in visitor subclasses throw "not implemented" exceptions.

Halfway through the semester, it was discovered that mock objects [2, Chapter 7] are very helpful in testing a compiler. A mock object is, as the name suggests, an object that pretends to be another object. We used jMock [10] (version 1.0) to mock subexpressions, e.g., the subexpressions of an `if` expression. A mock object can be told what methods will be invoked on it and what to return. This avoids having to test, for example, integer processing at the same time that `if` expressions are being tested; the mock object will indicate whether or not the right methods on the subexpressions have been called.

3.5 Things We Should Have Done

We did not do a good job judging some costs and benefits. For example, generating assembly code for the PowerPC is not trivial. It is perhaps a good target for students *really* interested in assembly code, but these students need to know that it comes at a cost—they may not get to an optimization step that others do. Targeting a virtual machine (like Parrot) is an easier target for the back end. *All* compiler courses face this same problem; however, it is greatly amplified during an interim.

Some acceptance testing would have been helpful to motivate changes in the compiler. Using a tool like FitNesse [6], we would enter a high-level acceptance test, describing a new feature for the compiler. The errors this test produces would indicate what unit tests to add in order to add abilities to our classes.

We both fell back on bad programming practices out of fear. One of us neglected his unit testing; the other let his schedule slip without readjustment. Discipline is the primary answer to these issues. Discipline would be even more important during interim when time pressures are keener.

Picking a good target for the generated code and the schedule slip actually have an issue in common: the ability to estimate coding time. Even if the estimates are way off, this practice would have forced us to think through the cost of generating PowerPC code, and we would be forced to readjust schedules.

In a class of ten to twenty students, it would also be possible to work on compilers in groups. Going at it (practically) alone was not a bad experience for us, but group work or pair programming or both would not change anything fundamental to the incremental approach described so far; it would just let more get done.

4. EVALUATION OF 15-IN-15

One significant drawback of this approach is the TDD knowledge that is needed. Using JUnit, FitNesse, and jMock is not trivial, and if students do not have previous experience with these technologies, time will have to be spent teaching them. This seems like a worthwhile trade-off, even at the cost of some compiler knowledge, because these are practical technologies used outside the classroom—turning the compiler course into a provably practical course! Having used these tools *heavily* on a *significant* project, students should not be embarrassed at the end of the class to list these technologies on their resumés.

Working on one compiler as a group would allow for more of the compiler to be written. If a group used pair programming and switched partners regularly, students should see and work with a significant portion of the compiler.

What/When	Traditional	Incremental, test-driven
Halfway through the semester.	Complete lexer and some parser for a complete language.	Complete front-to-back compiler for a simple (but non-trivial) subset of a language.
End of semester.	Complete lexer, complete parser, some static analysis, and little code generation for a complete language.	Complete front-to-back compiler for a significant portion of a language, including some optimizations.

Table 1: Average Student's Accomplishments

Shunning an up-front design gives both the instructor and the students the flexibility to change feature sets even on day fifteen. If a student discovers register allocation halfway through the interim, it can be added to the next feature set for the following day (of course, only after estimating the costs and benefits). Some care should be given, though, to the first few features sets for pedagogical reasons, so that the basics can be covered effectively and advanced features can be added easily.

Working on the whole compiler each week demonstrated very well why some language features are implemented where they are. For example, it quickly becomes clear that a parser has the abilities to handle operator precedence, and the benefits of this are seen immediately when the back end of the same compiler is implemented. Similarly, range checking of integers (for overflow) is most easily handled during static analysis when the integer can be evaluated directly; handling this in the lexer (in a regular expression) is not a fun way to spend an afternoon.

A spiral approach to teaching, which arises naturally with this incremental-development of a compiler, has the advantage of showing material to students multiple times. Their first exposure is shallow and very basic; subsequent exposures reinforce the basic material and introduce more advanced material. This repetition helps the students retain the material after the course is finished.

The other pedagogical benefit of our approach is the emphasis placed on the material (summarized in Table 1). One problem with a traditional front-to-back compiler course is that the early stages of a compiler are disproportionately emphasized more than the latter stages. When (not if) average students spend too much time finishing the tricky bits of their scanners and parsers, they fall behind and run out of time to finish the last stages, usually code generation. An incremental, test-driven approach forces the students to see *all* phases of their compilers *each* day. Each day may stress a different part of the compiler (e.g., arithmetic expressions stress the parser, `while` loops stress code generation). In the end, each stage of the compiler is stressed as much as it should be.

5. CONCLUSION

Two further projects are planned for Spring 2006. First, we plan to repeat our proof-of-concept experiment, this time developing ray tracers. Mock objects and acceptance tests will be used from the beginning to determine their actual effectiveness; we will estimate our coding times and plan more honest schedules. We are curious whether this approach works any better or worse educationally with ray tracers compared to compilers. Ray tracers are more modular, and we hope that this leads to some interesting possibilities.

Second, students in a programming-languages course will use incremental development to write an interpreter. Our programming languages course has significant coverage of regular expression and context-free grammars, so the interpreter project maps very well to our compiler experiment.

Our experience convinces us that incremental development with test-driven development is a great way to approach a front-to-back compiler in a compiler course. It provides great flexibility (even for a three week course); it exposes students to useful and practical tools; and it more fairly emphasizes course material. We have great hopes for our new projects and for this approach in general.

6. REFERENCES

[1] ANTLR-testing website. `http://antlr-testing.sourceforge.net`.

[2] D. Astels. *Test-Driven Development: A Practical Guide*. Prentice Hall PTR: Upper Saddle River, NJ, USA, 2003.

[3] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann: San Francisco, 2004.

[4] R. K. Dybvig, R. Hieb, and T. Butler. *Destination-driven code generation*. Indiana University Computer Science Department Technical Report #302, February 1990

[5] Eclipse website. `http://www.eclipse.org/`.

[6] FitNesse website. `http://www.fitnesse.org/`.

[7] D. Gallardo, E. Burnette, and R. McGovern. *Eclipse in Action*. Manning: Greenwich, 2003.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley: Reading, Massachusetts, 1995.

[9] S. Hoxey, F. Karim, B. Hay, and H. Warren, editors. *The PowerPC Compiler Writer's Guide*. Warthman Associates: Palo Alto, CA, 1996.

[10] jMock website. `http://www.jmock.org/`.

[11] JUnit website. `http://www.junit.org/`.

[12] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley: Reading, Massachusetts, 1999.

[13] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann: San Francisco, 1997.

[14] T. Parr. *ANTLR Reference Manual*, `http://www.antlr.org/doc/index.html`, 2005.