Design Choices in A Compiler Course

or

How To Make Undergraduates Love Formal Notation

Michael I. Schwartzbach
Department of Computer Science
University of Aarhus, Denmark
mis@brics.dk

Abstract

The undergraduate compiler course offers a unique opportunity to combine many aspects of the Computer Science curriculum. We discuss the many design choices that are available for the instructor and present the current compiler course at the University of Aarhus, the design of which displays at least some decisions that are unusual, novel, or just plain fun.

1 Introduction

The compiler course is an important component in the undergraduate Computer Science curriculum. Ideally, it ties together several aspects of the education, such as formal languages (regular and context-free languages, syntax-directed translation), programming languages (features and constructs), algorithms and data structures (ASTs, symbol tables, code selection, optimization), logic (type systems, static analysis), machine architecture (target platforms), and software engineering (phase slicing, versioning, testing). At the same time, the compiler project may involve the largest and most complex piece of software that the students so far have been required to handle.

Even though most compiler courses obviously have a common basic structure, a lecturer faces a host of design choices that must be made explicitly or implicitly. In the following such choices will be presented and discussed, both in general terms (based on an unscientific study of around 50 courses) and as they apply to the current compiler course at the University of Aarhus, the design of which displays at least some decisions that are unusual, novel, or just plain fun.

2 Design Choices

A compiler course must teach the students how compilers work. Beyond this obvious statement hides a multitude of different choices that influence the contents of the lectures, the style of the teaching, and the experiences of the students. Compiler courses may fill different roles in the curriculum, recruit students with different backgrounds, and focus on different aspects. Thus, the purpose of this section is not to describe an optimal point in the design space, but instead to make the many design choices explicit and to discuss their consequences.

Projects

Many compiler courses are focused on a compiler project where the students implement a working compiler of parts thereof. It is possible to keep the course purely theoretical, but it seems to be the consensus that learning-by-doing is eminently suited for this topic, and that implementing a compiler is a uniquely rewarding and empowering experience for students.

A project may be monolithic, meaning that the students build a complete compiler from scratch. While initially appealing, this approach is often too demanding and may strand students that get a bumpy start. At the other end of the scale, the course may be segmented into a series of unrelated minor projects or assignments that each are concerned with different aspects of the compilation process.

In between these extremes, many courses seek to provide a phase slicing where the students hand in different parts of the compiler at various deadlines. The phases may be implicit, or they can be made explicit by specifying strict APIs for their interfaces or by introducing a string of intermediate languages. This is more manageable, but fixed interfaces tend to limit the possible software designs while many intermediate languages tend to be confusing.

The compiler project is often large, time-consuming, and viewed as a rite of passage. Its solution is typically the largest and most complex piece of software that the students have written. Correspondingly, it is also a challenge for the teacher, since the project must be planned and tested in detail.

Source Language

The source language of a compiler is rarely a complete programming language, such as Java, C#, or Haskell. The many subtle details of such languages and their specifications are seen as distractions from the essential topics. Also, such languages typically contain many redundant features that increase the amount of grunt work in the project. Thus, most source languages are scaled-down versions of existing languages or invented for the occasion. There is a surprising fecundity, as a quick look through the hits in Google reveals the following source languages for compiler projects: Tiny, Cool, UnCool, MinC, MicroGCL, Tiger, Iota, PCAT, Tiny-C, SampleC, Lake, B-flat, DL07, Simple, Ada/CS, ice9, ALLCOT, F05, Z#, MiniCaml, MiniPascal, Pascalito, MiniOberon, SOOP, SIMP,

CSX, Tigris, Minila, C--, μOCCAM, MLPolyR, Dejlisp, and (Java seems to spawn the most imitators) Decaf, Irish Coffee, Espresso, TinyJava, MiniJava (several versions), MicroJava, Fjava, Javelet, StaticJava, CSX, j--, Jack, and Joos. Most of these languages include essentially the same features, but there is of course a fundamental distinction between functional and imperative languages. The huge variety may be viewed as an instance of the *Not-Invented-Here* syndrome, but it probably just reflects that most compiler teachers are having fun with their course and cannot resist the urge to tinker. Quite often the source languages exist in two or more versions, and the students may earn extra credit by implementing versions with extra features.

An entirely different approach is to use domain-specific source languages, which is probably how most students will potentially apply their compiler skills. In this case the students may even be allowed to design their own languages in response to challenges from various application domains. The advantage is that the project then includes a component of language design, but the main disadvantage is that the resulting languages may omit many important features (and that the teacher faces a Babylonic confusion).

Target Language

The choice of target language is less varied. Languages in the Java or C# families generally translate into the corresponding virtual machines (since JVM and .NET are quite accessible) and Pascal derivatives translate into a P-machine. Many courses choose assembly code as the target, generally x86, SPARC, or MIPS, sometimes in simplified forms such as SPIM. A third choice is to generate C-code, which seems to be a good match for domain-specific source languages. The choice of target language is also related to the discussion of frontends vs. backends in Section 5.

Implementation Language

The implementation language is generally one with which the students are already familiar, typically Java, C#, ML, or C++. The many Java-based source languages are almost always linked with Java as implementation language, which yields a certain elegance. Other courses explicitly use distinct source and implementation languages to increase the exposure to different languages.

That Extra Thing

Apart from the basics of compiler construction, a compiler course seems to have room for something extra. For example, the students may also acquire a detailed knowledge of functional programming if the source and implementation languages are both functional. Also, if the source language is domain-specific, then the students may leave the course with significant knowledge of the given application domain. As a final example, the compiler may be specified completely in logic programming or using an attribute grammar evaluation system,

in which case knowledge of the chosen formalism is added to the outcome of the course. This also relates to the discussion in Section 5.

This ability to use a compiler course as a vehicle for *That Extra Thing* is an important design choice of which teachers should be aware.

Specifications and Formalization

Compiler courses display a large difference in the level of formalization that is used in the specifications of languages, semantics, machines, and translations. To a large extent this reflects the pre-qualifications of the students and the preferences of the teacher. Most aspects of compiler technology may in fact be completely formalized, but this is rarely the style used, and it is certainly possible to be precise without being formal. As discussed in Section 6, selling the idea of useful formalizations may become a main purpose in the course.

The various components of the project are typically specified rather informally, using examples or prose. If the source language is a subset of a known language, then it may be specified as the original (often huge) language specification mentally projected onto the subset of the syntax that is allowed. This is actually a brittle technique, since programs in the subsyntax for subtle reasons are not always legal in the full language (as a stupid example, imagine allowing excluded keywords as identifiers). Reading full specifications of languages or target platforms is often a both harrowing and healthy experience for students.

Tools and Technology

Most phases of a compiler may be supported by specialized tools and, of course, entire compilers may be specified in compiler-generating frameworks. Apart from the occasional use of attribute grammar systems, compiler courses generally only automate scanning and parsing. There seems to be a certain air of conservatism in this respect, with Lex/Flex and Yacc/Bison still reigning supremely. Java and C# has specific and improved versions of such tools, as does in fact any major programming language with respect for itself. Generally, the more modern tools will offer better support, such as integrated scanners and parsers, automatic generation of AST code, and perhaps expressive power beyond LALR(1).

An alternative approach uses one-pass compilers, in the style of the classical Pascal compilers, with handwritten scanners and recursive-descent parsers. While this possesses some old-school charm, it misses out on important aspects (such as ASTs, phases, analysis, and optimizations) and it does not generalize to handle the complexities of modern languages.

Software Engineering

The compiler project is an excellent opportunity for gaining experience with software engineering techniques. The code is large enough to show how IDEs may help the programming task, and tools for code sharing and versioning

are relevant, particularly if the students work in groups. The architecture of the compiler may also be used to showcase advanced programming patterns as discussed in Section 7.

Skeleton Code

Most courses provide the students with a starting point in the form of a skeleton of the complete compiler. This helps the students structure the code and it makes their resulting compilers more uniform and thus easier to evaluate. The skeleton may be more or less detailed, which is a useful parameter for adjusting the work load.

An alternative strategy is to provide the students with a complete compiler for a different and often simpler source language, sometimes a subset of the source language for their project. This helps in providing a complete overview and enables the students to proceed by analogy.

Testing

Testing an incomplete compiler is a challenge. The generally recommended strategy is to output a version of the annotated AST after each phase and then manually inspect its contents. If the compiler is structured with explicit phases, a more ambitious testing harness may be possible, see Section 8. For the complete compiler it is useful to provide a test suite for diagnosing its correctness, and perhaps also for grading the project. In any case, it is important that students acquire techniques beyond black box testing of the completed compiler.

Documentation

A compiler course is certainly not a writing class. A program as large as the constructed compiler seems to call for extensive documentation, but most courses focus instead on writing code. Almost universally, the students are asked to comment their code (often using something like Javadoc) and to write two pages for each hand-in describing the structure of their code and the algorithms the employ.

Group Work

While a small or segmented project is suitable for an individual student, the larger projects typically allow or require project groups. The ideal size of a group seems to be three people, which corresponds well with other programming experiences. There a several benefits from project groups, including the direct training in collaboration, planning, and communication. The danger is of course that the learning outcomes may differ for the group members, in particular there is a danger of leaving a weaker member in charge of fetching sodas.

Exams and Grading

The project is generally the dominating activity on a compiler course, but this is not always reflected with a equal weight in the examinations. The final grade is typically a weighted sum of the individual project hand-ins, a number of midterms, and a final exam. The midterms and the final exam are of course individual, while the project hand-ins are often made in groups. For group projects the weight of the compiler code varies between 25% and 50% (with 40% being the median), and for individual projects it varies between 70% and 90% (with 70% being the median). In a few cases the weight of the project is zero, which seems to provide poor alignment as discussed in Section 11. The final exam is almost invariably a standard written test, focusing on the underlying theory.

The compiler code is evaluated by a combination of code inspection and functional testing. It is a huge advantage to enable automatic evaluation of test suites, which must of course be able to handle both positive and negative test cases, as discussed in Section 8.

Quite a few courses supplement the exam with an additional possibility for winning awards for such feats as "Best Compiler" or "Fastest Runtime". Contests like these are a surprisingly effective way of spurring on the better students. A winner's cap or t-shirt (or merely a mention on a Web page) is apparently ample payment for those extra 100 hours of work.

3 The dOvs Course

The Department of Computer Science at the University of Aarhus has an undergraduate compiler course (called *dOvs*) in the final year of the B.Sc. program. It has the unique advantage of being a mandatory course, meaning that it is attended by around 80 students every year. Following an earlier run in 1997-2001, the course was redesigned in 2005 based on the past experiences and with explicit attention to the many design choices presented above.

4 A Case for Large Languages

As mentioned earlier, most source languages for compiler projects are smallish. The motivation is that the students have a limited time for their project and that a small language allows them to focus on the important concepts.

In contrast, the source language for the dOvs compiler project is huge chunk of Java. The largest language we consider is called Joos2, and it corresponds to Java 1.3 with the omission of package private declarations, some control structures, and various odds and ends. Conceptually, it is every bit as complicated as the full language. The Joos2 language is the target for students going for extra credit. The basic requirement is the Joos1 language, which further omits instance and static initializers, multi-dimensional arrays, interfaces, and simplifies local initializers and method overloading.

The benefits of using a large language is clearly that the students obtain a realistic insight in to the workings of a full compiler that they have been using for their entire previous studies. Their sense of achievement and empowerment clearly grows with the street credibility of the source language. Also, by considering a full modern programming language, the students will encounter many concepts and challenges that are absent in smaller projects, as discussed in Section 5 and in Section 6.

The challenge of a large language is of course to make the projects succeed in the given time frame. Clearly, this requires that the starting point is a large skeleton compiler. Our model implementation of the Joos2 compiler is 12,207 lines of code (hand-written that is—SableCC generates a further 64,290 lines). The skeleton that is provided is 8,466 lines of code. The remaining 3,741 lines of code must then be written by the project groups (but generally they require between 5,000 and 10,000 lines to fulfill this task). The skeleton code includes 3,000 lines of code for defining Java bytecodes and the peephole optimizer discussed in Section 10. This leaves around 5,000 lines of real compiler code that the students are given for free. Clearly, the danger is that they could miss important points by not fully understanding this code. However, it seems impossible to complete the skeleton without having a detailed understanding of its inner workings.

The students are provided with a complete compiler for the Joos0 language, which is a tiny static subset of Joos1 that corresponds to many of the source languages used for other compiler courses. For completeness, a lecture is used on the implementation of a narrow, one-pass compiler for a subset of C that is compiled into the IJVM architecture (in 841 lines of code).

5 Frontend, Backend, or Middle-End?

Compiler courses can roughly be divided into two categories: those that are *frontend-heavy* and those that are *backend-heavy* (with only a few large courses managing to be both at the same time).

The first category spends a large part of the course on formal languages, in particular on finite automata and context-free parsing. The construction and correctness of LALR(1) parsing tables is still a cornerstone of many courses, sometimes consuming half the available time. Of course, LALR(1) parser generators are still the most common choice, even if many other alternatives are available, including general parsers that can handle all unambiguous languages. These compiler courses often double as formal languages courses, but even in this setting LALR(1) parsing theory is probably not the most vital topic.

The second category spends a large part of the course on code generation, register allocation, and optimization for a realistic processor architecture. In this setting, the project source language is often a simple static subset of Java or C that allows the frontend to be dealt with as painlessly as possible.

Wedged between the traditional frontend and backend is the *middle-end* which deals with the semantic analysis, including symbol tables and type check-

ing, which the majority of compiler courses dispenses with in a single week. This actually seems paradoxical, since e.g. the vast majority of the Java and C# language specifications deal with exactly this topic. The dOvs course has been designed to be middle-end-heavy.

For the frontend, dOvs has the advantage of following a mandatory course on formal languages, so little time needs to be spent on the basic definitions. The students learn how LALR(1) tables work, but not how they are constructed. This provides a sufficient basis for understanding LALR(1) conflicts and reading error messages from SableCC. The students are provided with a SableCC specification for the Joos1 language, but with a dozen missing features that must then be added (an the resulting LALR(1) conflicts must be resolved).

The backend is naturally light, since our compiler generates Java bytecode and the JVM has a simple architecture. Even so, optimization does play a major role in the course, as discussed in Section 10.

But the majority of the course deals with the voluminous middle-end of a Java compiler: weeding of ASTs for non-LALR(1) syntactic restrictions, building the global class library, building environments, linking identifiers to declarations, resolving import declarations, checking well-formedness of the global class hierarchy, disambiguating compound names, type checking, making implicit coercions manifest, constant folding, and performing static analyses for reachability and definite assignments. Such tasks are the main challenges of modern compilers and, consequently, it seems reasonable to give them a proportional amount of attention.

6 Learning to Love Formal Notation

The dOvs course is designed with the clear goal that the students should learn to love formal notation. This is not a trivial task, since many students start out with the exact opposite attitude. But it is generally the case that a tool is best appreciated when it is desperately needed.

The Joos languages are formally defined by a combination of its syntax, the Java Language Specification (JSL), and a list of excluded features. As mentioned earlier, it is actually a subtle task to ensure that every Joos1 program is also a legal Joos2 program, and that every Joos2 program is also a legal Java 1.3 program.

The JLS is a formidable document that is heavy reading for anyone, in particular for undergraduate students. Thus, the lectures present compact formalized explanations of the difficult aspects of the Joos (and Java) semantics. The lectures (and the accompanying 600 slides) use appropriate formal notation to explain the sometimes convoluted semantics of Java. This happens at a time when the students are highly motivated, since they are trying to implement that semantics at the same time, and they quickly notice that the formal notation is often a direct guide to the code that must be written.

Various formalisms are in play here. The well-formedness of a class hierarchy is defined by sets and relations (DECLARE, INHERIT, and REPLACE) that are

populated through inductive rules and subjected to constraints phrased in first-order logic. Static type checking is defined through ordinary inference rules, with inductively defined relations as side conditions. The static analyses for reachability and definite assignment are defined using least solutions for set constraints. Code generation is specified as a syntax-directed translation based on templates. Finally, peephole optimization is presented as the fixed-point closure of a transition relation.

As an illustration why the practicality of formal notation becomes clear, consider the rules for definite assignments. In the JSL these are defined in 474 lines of prose of the following poetic form:

> The definite unassignment analysis of loop statements raises a special problem. Consider the statement while (e) S. In order to determine whether V is definitely unassigned within some subexpression of e, we need to determine whether V is definitely unassigned before e. One might argue, by analogy with the rule for definite assignment, that V is definitely unassigned before e iff it is definitely unassigned before the while statement. However, such a rule is inadequate for our purposes. If e evaluates to true, the statement S will be executed. Later, if V is assigned by S, then in the following iteration(s) V will have already been assigned when e is evaluated. Under the rule suggested above, it would be possible to assign V multiple times, which is exactly what we have sought to avoid by introducing these rules. A revised rule would be: V is definitely unassigned before e iff it is definitely unassigned before the while statement and definitely unassigned after S. However, when we formulate the rule for S, we find: V is definitely unassigned before S iff it is definitely unassigned after e when true. This leads to a circularity. In effect, V is definitely unassigned before the loop condition e only if it is unassigned after the loop as a whole! We break this vicious circle using a hypothetical analysis of the loop condition and body. For example, if we assume that V is definitely unassigned before e (regardless of whether V really is definitely unassigned before e), and can then prove that V was definitely unassigned after e then we know that e does not assign V.

In the formal notation, the definite assignment analysis is phrased in 7 slides with set constraints of the following form (that directly translates into code using a `DepthFirstAdapter` visitor pattern from SableCC):

$$
\begin{array}{l}
\{\sigma\ \ x = E\texttt{;}\ S\}: \\
\quad B[\![E]\!] = B[\![\{\sigma\ \ x = E\texttt{;}\ S\}]\!] \\
\quad B[\![S]\!] = A[\![E]\!] \cup \{x\} \\
\quad A[\![\{\sigma\ \ x = E\texttt{;}\ S\}]\!] = A[\![S]\!] \\
\texttt{while (}E\texttt{)}S: \\
\quad B[\![E]\!] = B[\![\texttt{while (}E\texttt{)}S]\!] \\
\quad B[\![S]\!] = A_t[\![E]\!] \\
\quad A[\![\texttt{while (}E\texttt{)}S]\!] = A_f[\![E]\!]
\end{array}
$$

Thus, the students experience that the formal notation is their friend that enables them to meet their deadlines. Hopefully, this will change many skeptical attitudes.

7 Explicit Phases Through SableCC and AspectJ

As mentioned, there are countless tools for generating scanners and parsers, and still many even if they are required to be compatible with Java. We have chosen to use SableCC, which is of course to some degree a matter of taste, though it does provide most modern conveniences: integrated scanner and parser, automatic parse tree construction, and visitor patterns for tree traversals (in fact, we use a souped-up version of SableCC with more powerful features for tree navigation). However, there are more objective reasons why we feel SableCC is uniquely suited for a compiler course.

SableCC allows the specification of ASTs in a separate grammar that is really just a recursive datatype. It is then possible to specify syntax-directed translations from concrete to abstract syntax trees. Apart from being convenient, this teaches the students about inductive translations in a practical setting. This feature is also used to illustrate desugaring, by translating `for`-loops into `while`-loops during parsing.

Phase slicing can be taken to an extreme length by combining the ASTs of SableCC with simple features of AspectJ (which is the real implementation language, though the students hardly notice this). A compiler phase needs to perform one or more traversals of the AST but also to decorate the AST nodes with additional information, such as symbol environments and types. Generally this means that the class definitions for AST nodes must be extended with phase-specific fields, which poses several problems. First, the actual AST node classes are autogenerated by SableCC, so it is inconvenient and dangerous manually to extend them. Second, if the phases are considered one at a time, then it requires an awkward prescience to declare these extra AST node fields in advance. Third, the specification of a given phase will be scattered over several files, which is an inconvenient software architecture.

Using AspectJ, extra fields can be injected into the AST nodes using inter-type declarations. For example, the skeleton code for the type checking phase starts as follows:

```
public aspect TypeChecking extends DepthFirstAdapter {
  /** The static type of the expression */
  public PType PExp.type;

  /** The static type of the lvalue */
  public PType PLvalue.type;

  /** The declaration of the field referenced in this lvalue */
  public AFieldDecl AStaticFieldLvalue.field_decl;

  /** The declaration of the field referenced in this lvalue */
  public AFieldDecl ANonstaticFieldLvalue.field_decl;

  /** The declaration of the method invoked by this expression */
  public AMethodDecl AStaticInvokeExp.method_decl;

  /** The declaration of the method invoked by this expression */
  public AMethodDecl ANonstaticInvokeExp.method_decl;

  /** The declaration of the constructor invoked by this expression */
  public AConstructorDecl ANewExp.constructor_decl;

  /** The declaration of the constructor invoked by this statement */
  public AConstructorDecl ASuperStm.constructor_decl;

  /** The declaration of the constructor invoked by this statement */
  public AConstructorDecl AThisStm.constructor_decl;

...

}
```

Here, several autogenerated AST node classes are extended with extra fields for information synthesized by the type checker. Using this technique, all concerns of the type checker is collected in a single file, and the autogenerated code can

safely be extended.

8 Unit Testing Through Phase Mixing

Testing a compiler during development is a difficult challenge, since only a complete compiler has a functional behavior. The students are encouraged to program an AST pretty-printer that after each new phase is extended to also print the newly added AST decorations. This simple technique goes a long way, but we can do better.

The use of AspectJ means that each phase resides in a single file. The use of SableCC with syntax-directed construction of ASTs mean that the interface between phases is fixed. This combination means that the phases of two Joos compilers may literally be mixed to produce a new hybrid compiler. We exploit this property by providing a complete and correct model implementation with which the students may build and test hybrid compilers.

Assume that a group is working on the type checker. To perform a functional test of this phase, they may build a hybrid compiler consisting of the model compiler with only the type checking phase substituted by their own. To check the allround progress, they may build a hybrid compiler consisting of their own phases up to an including the type checking phase mixed with the remaining phases from the model compiler. The students must of course not be allowed access to even the class files of the model compiler (since Java is vulnerable to decompilation), so the building and testing of a model compiler is performed by submitting phases to a Web service.

The testing of a compiler is quite extensive. In a full test the compiler is exposed to a test suite of 1,149 Java programs that each test a tiny specific property. This collection has been constructed over the three years this course have run, with new test programs being added whenever another potential kind of error is discovered. A simple positive test looks as follows:

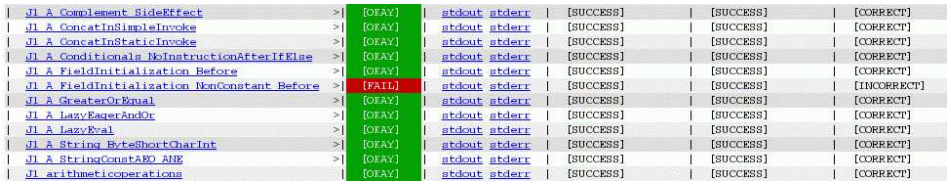
```
// TYPE_CHECKING
public class J1_constructoroverloading {
  public int x = 0;
  public J1_constructoroverloading() {
    this.x = 23;
  }
  public J1_constructoroverloading(int x) {
    this.x = x;
  }
  public static int test() {
    J1_constructoroverloading obj1 = new J1_constructoroverloading();
    J1_constructoroverloading obj2 = new J1_constructoroverloading(100);
    return obj1.x + obj2.x;
  }
}
```

By convention, correct runs will always return the value 123. A typical negative test looks like:

```
// JOOS1: PARSER_WEEDER,JOOS1_THIS_CALL,PARSER_EXCEPTION
// JOOS2: TYPE_CHECKING,CIRCULAR_CONSTRUCTOR_INVOCATION
public class Je_16_Circularity_4_Rhoshaped {
  public Je_16_Circularity_4_Rhoshaped() {
    this(1);
  }
  public Je_16_Circularity_4_Rhoshaped(int x) {
    this(1,2);
  }
  public Je_16_Circularity_4_Rhoshaped(int x, int y) {
    this(1);
  }
  public static int test() {
    return 123;
  }
}
```

It must generate the kind of error that is mentioned in the comments for respectively Joos1 and Joos2.

In general a test program may produce many different status values, depending on the success or failure of the compilation, the assembling, the class loading, the runtime, and the output. A snippet of the output from the test driver looks as follows:



The entries in the table are links that display all further details. In the first year, the test driver ran directly on a web server which was run to the ground as it quickly turned out that the students became addicted to using it. Subsequently we have implemented automatic filtering so only those tests relevant to the submitted phases are used, and the test driver now uses a farm of 17 dedicated test servers.

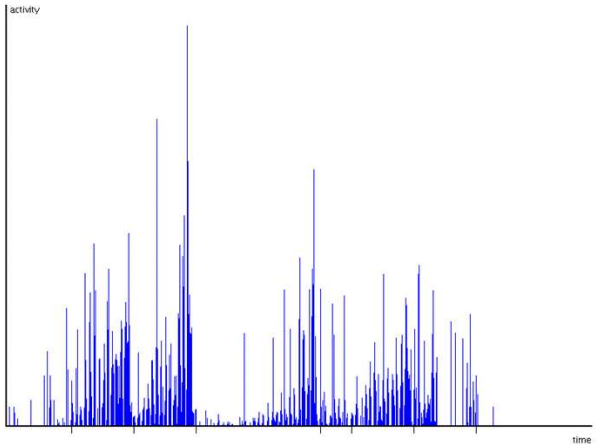
Another advantage of phase mixing is that students failing to complete a given phase may still continue the project by relying on the corresponding phase from the model compiler.

9 Incremental Feedback

The students receive extensive feedback during the course. The online test driver provides a continual evaluation of their code. We have a webboard staffed by teaching assistants, where 12% of the questions receive replies within 5 minutes, 42% within 1 hour, and 94% within 12 hours. Also, each group has a 30 minute weekly consultation with a teaching assistant. Finally, the groups must maintain a documentation blog, where they also receive feedback.

We also monitor the students closely. All activity in the system is logged and used for various statistics. A primary one is the activity curve, which shows how

hard the students are working as a function of time (measured as a weighted sum of the logged activities). Each year has shown exactly the same pattern, which looks as follows:



There are of course some marked spikes for each deadline, but overall the work load has a reasonable distribution.

The project is evaluated through points that are awarded for each phase. These are broken down into tiny individual components in the range between 0 and 2 points, which ensures a uniform evaluation. The students can see these points on a group homepage as soon as they have been awarded. Chasing points becomes something of an obsession.

We also maintain a visual presentation of how close the groups are to completing the next hand-in:

There is one horizontal line for each group showing the proportion between test programs passed (green) and failed (red), sorted by success rate for effect and anonymity. The collected pictures are also stored in 10-minute snapshots as a fascinating movie, showing the struggle between red and green as the groups work towards the next deadline.

The extensive logging is also useful to prevent cheating. In the few cases we have experienced, suspicions of code similarity were easily confirmed by observing anomalous behaviors in the log files.

10 The Peephole Contest

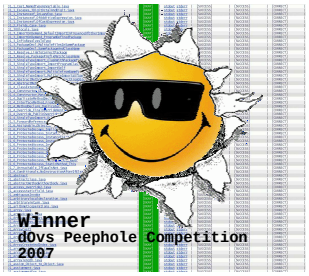
One hand-in deals with peephole optimization of Java bytecode. The syntax-directed code generation often produces naive code that is ripe for optimization.

We have developed a domain-specific language for specifying such peephole patterns, one of which may look as follows:

```
pattern p25 x: //comparison_in_ifcmp2
  x ~ ifcmp (e0, l0)
      ldc_int (i0)
      goto (l1)
      label (l2)
      ldc_int (i1)
      label (l3)
      if (e1, l4)
   && i0 == 0
   && i1 != 0
   && l0 == l2
   && l1 == l3
   && e1 == eq
   && degree l2 == 1
 -> 4 ifcmp (negate e0, l4)
      ldc_int (i1)
```

The compiler then contains a peephole engine that will apply all such patterns on the generated code until no pattern is applicable.

The students are invited to compete in creating the most effective collection of peephole patterns, measured in the total size of the bytecode generated for a benchmark suite. The winners receive a highly coveted t-shirt:



The bar is set fairly high, as the winning group generally produces hundreds of sophisticated patterns to secure their position. Without the competition, it is unlikely that this extra effort could be mobilized (at the last stage of the project).

11 Exams and Grading

The projects are evaluated on a scale between 0% and 110% (including extra credit). The highest ever score so far is 107%. The project is weighted with 70% in the final grade, which is large considering that the project is done in a group. However, the principle of *alignment* dictates that the exam should reward the activity which best promotes learning, and this is clearly the project work.

To ensure individual grades, the course concludes with a 75 minute multiple-choice test (allowing partial knowledge) that covers the basic theory and details about the project. Multiple-choice tests are unfairly viewed as being superficial, but the questions may in fact be quite deep:

Consider the method invocation `A.B(1,2,3)`. To which category can `A` *not* belong?

- *a* ☐ A class name.
- *b* ☐ A static field name.
- *c* ☐ A non-static field name.
- *d* ☐ A local name.
- *e* ☐ A package name.
- *f* ☐ A formal name.

Clearly, superficial knowledge is not enough to answer such questions. The multiple choice test often yields a final difference of one to two grades among group members, and it invariably rewards those students that the teaching assistants predict to be the best.

The students generally do well, and many receive their highest grade in their degree program in this course:

12 Conclusion and Acknowledgements

Compiler courses are important and have been taught for a long time. We have identified many design choices that are available to teachers and have discussed some of their consequences.

The dOvs course has been designed with explicit consideration of these choices and with the goal of being novel and fun. The main characteristic of the course is that the project is huge and complicated, forcing the students to appreciate software engineering techniques and to grow to depend on formal notation as a guide to express the semantics of the source language in the implementation.

The course and its extensive infrastructure has been developed and implemented in close collaboration with Aske Simon Christensen, Janus Dam Nielsen, and Johnni Winther.