Compiler Construction as an Effective Application to Teach Object-Oriented Programming

Akim Demaille Roland Levillain*
EPITA Research and Development Laboratory (LRDE)
14-16, rue Voltaire; FR-94276 Le Kremlin-Bicêtre; France

Abstract

Compiler construction: a course feared by most students, and a competence seldom needed in the industry. Yet we claim that compiler construction is a wonderful subject that benefits from virtually all the computer-science topics. In this paper we show in particular why Compiler Construction (CC) is a killer example for Object-Oriented Programming (OOP), providing a unique opportunity for students to understand what it is, what it can be used for, and how it works.

1 Introduction

The curriculum at EPITA, a French private computer science university, emphasizes practical implementation projects. Ten years ago the decision was made to assign a large scale project. The requirements included OOP, design patterns [8], best software engineering practices (including tests, documentation, and version management), C++ and teamwork management. Compiler Construction (CC) provided a nice application topic, hence the inception of the Tiger compiler project [3].

Over three (optionally five) months, while they are taught C++, undergraduate students implement stage after stage a compiler in C++ for an Object-Oriented Language (OOL): Tiger [1, 5]. The CC lecture and the project courses are integrated. Since "students will (most likely) never design a compiler" [2] CC is *not* the primary goal. The teaching objectives are focused on the process, the "how", not the "what."

Therefore rarely used tools or languages that are tailored for CC are not used. The compiler itself is as simple as possible. The core assignment produces Intermediate Representations (IRs); skillful students are encouraged to work till completion, producing assembly language. See [3, 4] for more details.

Claims and Outline Based on our ten-year experience, we claim that carefully crafted CC is an OOP killer example. Deep understanding of compiler is not required for this paper, Figure 1 provides a crash course. Section 2 presents four features of compilers that make them unique to demonstrate OOP. Section 3 explains how the course and project are bound. In Section 4 we provide additional support for our claim, including feedback from the students. Finally, we conclude in Section 5.

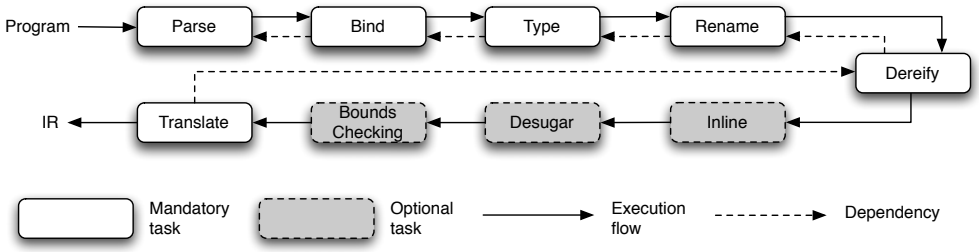
2 The OO Crossing

OOP is surprisingly adequate to address some of the data and algorithms used by compilers (Section 2.1). At a higher level, the global architecture of a compiler and its components fully benefit from object-orientation (Section 2.2). Of course a compiler for an OOL is particularly suited to help understanding what's OOP (Section 2.3). And finally, students get an in-depth knowledge of the OOL they implemented their compiler in (Section 2.4).

2.1 Data and Algorithms

Compilers use a large number of data structures and algorithms, which, of course, benefit from OOP. More importantly, some of them provide an excellent introduction to the design of a class hierarchy, and others require tortuous implementations if not handled via OOP.

*Also with Université Paris-Est, LABINFO-IGM, UMR CNRS 8049, A2SI-ESIEE, Cité Descartes, BP 99, FR-93162 Noisy-le-Grand Cedex, France



A compiler typically features three parts: the *front-end* analyzes the input program, the *middle-end* is independent of the source and target languages, and the *back-end* generates the output program in the target language. This figure focuses on the front-end.

First, the source is *parsed*, resulting in an *Abstract Syntax Tree (AST)*. The AST is the main data structure of the front-end; it is analyzed/annotated/modified by the various modules. Then the *bind* task implements scoped name analysis: it matches identifier uses with their declaration. The *types* are checked. *Renaming* all the identifiers so that they are unique simplifies later transformations. Object Oriented (OO) structures are translated into object-less equivalent constructs at the *dereify* stage. *Function inlining* is an optimization, *desugaring* translates high-level constructs into more primitive ones, and *bound-checking* instruments the code to catch out-of-bounds array accesses. Finally, the front-end *translates* into an Intermediate Representation (IR), passed to the middle-end.

Figure 1: Crash course on the Tiger compiler front end.

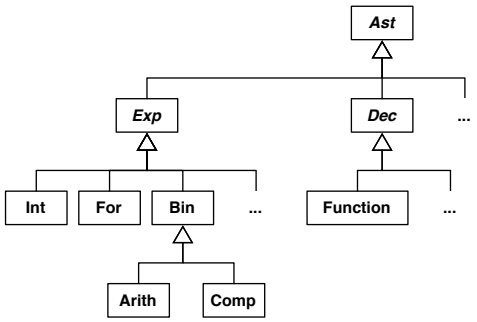


Figure 2: AST hierarchy.

2.1.1 Abstract Syntax Tree

The Composite design pattern is probably the most emblematic example of OOP: what can be a simpler introduction to hierarchies than the one of an AST?

Source languages are described thanks to a grammar composed of *terminal symbols* (also called *tokens*) such as `INTEGER` or '`+`', and of *non-terminal symbols* such as `expression` or `statement`. A *grammar* is a set of *rules* that define the relations between the symbols, for instance:

```
/* Exp */ exp:
  /* Int */ INTEGER
| /* Add */ exp "+" exp
| /* For */ "for" IDENTIFIER ":=" exp "to" exp "do" exp
| /* Many more rules.  */
```

To each syntactic symbol corresponds a class of the AST, and *the hierarchy directly reflects the grammar* rules (see Figure 2). In the above example, the classes `Int`, `Add` and `For` all derive from the `Exp` class. The distinction between abstract and concrete classes arises naturally: `Exp` denotes a disjunction of cases, but "does not exist by itself".

Student can be driven to provide smarter hierarchies by considering the properties of the symbols. The handling of the binary operators (arithmetic, order, equality) shows this very well. They all have two operators, which should be factored in a `Bin`(ary) class. From the type checking point of view, there are two groups: those that apply to integers and strings (order and equality) and those that apply on numbers only. This can be reflected by two additional abstract classes, `Arith`(metic) and `Comp`(arison).

Even with the simplest processing of an AST—because there can be many classes—students quickly spot redundancies and factor. Many invent the Template Method design-pattern when implementing the AST pretty-printer: all the `Bin` descendants first display a '`(`'[1], then the left-hand side, the operator, the right-hand side and finally '`)`'. The general template is implemented in the `Bin` class, and the specific part (the operator) is delegated to the sub-classes.

[1] Without the parentheses '`(1+2)*3`' would be displayed as '`1+2*3`'. '`((1+2)*3)`' is longer than needed, but correct.

2.1.2 Translation To Intermediate Representation

The final stage of the front end, `Translator`, translates the AST into some IR used by the middle end. It exhibits an interesting problem which is very cleanly handled by OOP. The translation of some constructs depends on their *use*. For instance `a < b` used as a statement can be simply discarded (think of '`(void) a < b;`' in C), used as an expression it should be normalized to `0` or `1`, and used as a conditional in an `if` statement, it naturally translates into a conditional branching instruction. Traditional traversals of the AST cannot cope nicely with the way a node is *used*: it requires that the processing of a child knows the type of its parent (and in some cases, of its ancestors!).

Appel [1] suggests a beautiful OOP implementation: translation nutshells. Instead of returning the completed translation of a node, convert each AST construct into a proto-translation, a nutshell whose type depends on its actual nature. There are three proto-translation types: `Expression` (values involved in computations), `Statement` (expressions whose values are discarded, loops, etc.), and `Comparison` (expressions well-suited for branching). Then a parent completes the translation of a child by calling a method that depends on its need: `as_statement`, `as_expression`, and `as_condition`.

2.1.3 Language Extensions

We submit challenging optional tasks to keep skillful students interested. Language extensions (overloading support, additional primitive types, etc.) are a good topic, but they require that base stages be extended to support the new constructs. This is cleanly handled by OOP: sub-class the visitors (e.g., derive an `OverloadTyper` from the `Typer` to compute the types of overloaded routines).

2.2 Architecture and Design Patterns

Although compilers were amongst the most complex pieces of software 50 years ago, today the bird-eye view of their architecture is usually surprisingly simple: a sequence of transformations[2] (or *tasks*).

[2]These transformations can be complex and in the case of optimizing compilers their scheduling is complex; this is not the case in a pedagogic compiler.

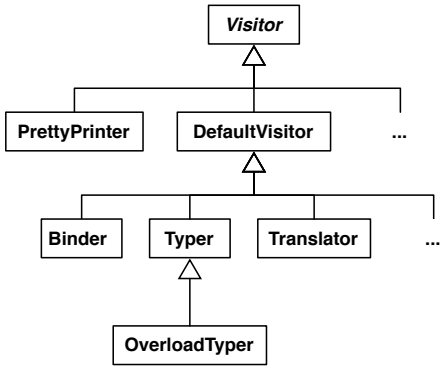


Figure 3: Visitor hierarchy.

All the stages in the front end apply on the AST. Instead of continuously editing the AST classes, the VISITOR design pattern is used. This pattern is a convenient implementation of traversals that decouples the algorithm (the traversal) from the data (the AST). The relevance of this pattern is perfectly well understood by the students because they deliver milestones of their compiler: at each new stage we provide new components (e.g., visitors with gaps) which they simply *add* to their project. Using classical virtual methods instead or VISITORS would *prevent* modularity, since they require students to alter the base classes, that they may already have adjusted to their needs, to add new behaviors.

Common parts in the visitors, starting with the traversal itself, provide a nice impetus for more OOP: introduce a visitor hierarchy to factor common parts (see Figure 3).

The whole compiler is "customizable", supporting 60 modules and a large variety of different configurations, including optional assignments (see Figure 1). The explicit sequencing between the different tasks would be cumbersome to write, a maintenance nightmare. The different stages of the compiler are therefore expressed as `Task` objects that declare their dependencies (dashed arrows in Figure 1) to the scheduler which computes the effective sequence (solid arrows). This is an instance of the COMMAND design pattern.

Other design patterns fit compilers, too many to enumerate. Of course SINGLETONS are... numerous (e.g., memory pool). The FLYWEIGHT pattern factors occurrences of an identifier to a single one. Not only does it save space, but it also dramatically speeds up the handling of identifiers: string operations (hashing, equal-

ity, order) become single CPU instructions on pointers.

2.3 Object-Oriented Source Language

Students get special insight with OOP when implementing an OOL.

2.3.1 Understanding the Semantics

In our experience, most students quickly understand what OOP is: encapsulation, data hiding, inheritance and polymorphism. They also are quickly able to make profit of object-orientation in most situations. Yet, most lack a precise and formal view of what an OOL is. Students who read language standards are rare (and should be hired immediately), usually they content themselves from what the courses and the course book provide: more of a tutorial than a reference guide.

Studying a *simple* language, with well defined semantics, allows the teacher to focus on the key components of OOL, to emphasize the dark corners (such as covariance vs. contravariance issues), or even to highlight design choices made invisible by the syntax (the dot notation, '`obj.fun(arg)`', hides the fact that polymorphism is usually restrained to the first argument, '`obj`').

Believing that we understand specifications is one thing, implementing them is another.

2.3.2 Understanding the Dynamics

The best means to feel the "Ah ah!" effect on OOP is to implement it.

Sticking to its guideline, the Tiger project makes sure that students' attention is not disrupted from the topic, OOP, by devoting a well defined and isolated task to it: "de-reification". This task transforms an object-oriented Tiger program into a plain Tiger project. Because it works on the high-level language instead of some low-level machine code, students of every skill can understand it. Because no other processing is performed, the attention is fully devoted to the mechanism of OOP. In addition, it is the opportunity to present different implementation of OOP, based of vtables as in C++, or on dispatching functions [13].

Thanks to this insight students can bring OOP into non-object-oriented languages (when they are not given the choice on the implementation language). Yet they are still able to explain why support from the compiler, rather than a library, is a better option.

2.4 Object-Oriented Implementation Language

Finally, as a piece of software, the compiler must be written in some programming language. Obviously, since there are many opportunities for object-oriented design in a compiler (see Section 2.1 and 2.2), an OOL is well-suited. But which one? We chose C++ over more recent languages like Java or C# for a number of reasons.

Rigorous and Demanding As Stroustrup [11] puts it himself "C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off". As any sharp tool, C++ can provide excellent results when mastered, or return itself against a lazy programmer. This is not the case of some "too comfortable languages" such as Java [7].

Multi-paradigm C++ supports a wide spectrum of programming styles. For instance, we teach different memory management techniques: Resource Acquisition Is Initialization (RAII) (a C++ idiom), flyweight pool, reference counting. This is impossible in garbage-collected languages. Generic Programming (GP) is heavily used: generic containers and generic algorithms are numerous. Parts use Functional Programming (FP), and even meta-programming: programs run at compile-time [4, *Goals* Sections].

Existing Libraries C++ often looks poor compared to other OOL when it comes to its standard library, STL [10]. For instance it does not even provide hash tables! But using stock C++ today would be weird: knowing and using Boost [12] is part of the C++ curriculum. Boost libraries are excellent material for teaching OOP, GP, and FP: these peer-reviewed libraries enlighten good C++ practices, design patterns, code factoring and reusing, as well as portability.

3 Course and Project

Managing such a project, spanning three to five months, with more than two hundred and fifty students per class, i.e., from about four hundred compiler deliveries (the project is made in

groups of 4 students, with 6 deliveries), is a challenge in itself.

Organization The Tiger team is composed of (i) one teacher who manages the project, writes the assignments and examinations, maintains the reference compiler, (ii) some co-maintainers who contribute mandatory or optional parts of the compiler, (iii) about twenty *assistants*, students of the previous class willing to assist younger students. They are also in charge of oral examinations.

We use Web sites to publish information, mailing lists to coordinate ourselves, and public newsgroups for asynchronous discussion with the students. A 200+ page document [4] details all the assignments and makes explicit what students are expected to learn. It also proposes ideas of optional extensions for the most enthusiastic students.

The project starts after students have completed a "Languages, Automata, Grammars, Parsing" course. To learn the language they first have to write a significant program in Tiger. They will use this assignment, together with other provided big programs, to exercise their compilers. Almost concurrently they start writing the scanner and parser. At the same period C++ lectures start.

One Milestone Assignment and support code for a stage are typically published on a Monday, and the delivery is two to three weeks afterward. In the meanwhile, the assistants are available at almost any time. On the Friday at noon, students must have uploaded their package on the Web site. It launches the automated testing procedure, which evaluates every stage of the compiler. It computes a global correction grade, giving higher penalties to errors in earlier phases. The result is published to the members of the group who may decide to "re-upload" within two days, which causes a 10% penalty. The following week the oral examination is organized. One assistant spends about 45mn with a group, asking questions, evaluating answers, browsing the code, and helping students to understand and overcome their problems.

Tools In such a large environment, automation is salvation. Many auxiliary tools were written [6] to ensure via a daily build that the reference compiler suffers no regression, to check the students packages, to run the test suites, to generate the documentation samples, to generate the AST, to produce the code delivered to student etc. Many of these tools are free software, available from the project's Web site [9].

4 Discussion

We conducted a survey at the end of the year among 204 students, to ask them which notions they thought were best taught by the compiler assignment. The item with the best mark (3.58/5) was "understanding how programming languages work", followed by "object-oriented modeling and design patterns" (3.18/5) and "C++" (3.07/5), which all three are actually the main goals aimed at by the project. However, we were surprised to find that *programming languages* were felt as better taught than C++, as we put the emphasis on the latter item rather than on the former during courses and the project. We think that using a more complex OO source language to be compiled actually helped the students to apply what they had learned, and better understand how object-oriented languages work. This idea is indeed confirmed by the fact that students seemed to have overcome most of the difficulties introduced by the OO constructs in the front-end: according to the survey, the parts considered as the less difficult ones were "name binding" and "type-checking" – the ones that were actually the most affected by the OO extension.

Over the years the project became easier, and less ambitious [3]. Infrastructure is also provided for highly skilled students. As an unexpected result, proficient students had lost some interest in the project: they felt the pleasure of writing a compiler was stolen from them. To meet their expectations, each stage now offers a set of options. They range from easy to ambitious and provide more insight into CC (which is not the purpose of the core assignment). Thanks to this infrastructure, new refinements were proposed, and more students implemented options.

As the project is long and teaches many subjects, it is usually perceived by students as having a strong impact on their schooling. During yearly debriefings, many of them report that the project was demanding. However we have received good reports from alumni each year about the project, sometimes several years after their graduation! Frequent remarks include the actual effectiveness of the project at teach-

ing design patterns, and that they are currently using them in their daily work. Many of them report that they have gained a status of expert w.r.t. C++, design, development process and tools among their team. And sometimes they even admit they reused a technique from the Tiger Compiler in their current project, almost *as-is*! Though informal, we rate these "success stories" as a good indicators of the relevance of the project at teaching modern and accurate OO and C++ design and programming.

5 Conclusion

In this paper, we have reported a ten-year experience on how Compiler Construction can be profitable to teach OOP to computer science students. Key ideas include the use of OO principles both as a tool (the compiler given as assignment) and as an object of study (the source language of this compiler); assignments with many steps and code with gaps; and a good toolset to help students understand and develop their compiler. We believe that good examples of pedagogical OO assignments must exhibit advanced, carefully crafted material: C++ and modern tools and idioms have proved to be very valuable assets to that intent. Our Tiger compiler assignment has changed over the years to follow the evolution of our OO course, and will continue to serve as one of our major pedagogical platform in the forthcoming years.

Acknowledgments We thank Alexandre Duret-Lutz for his proofreading and comments.

References

[1] Andrew W. Appel. *Modern Compiler Implementation in C, Java, ML*. Cambridge University Press, 1998.

[2] Saumya Debray. Making compiler design relevant for students who will (most likely) never design a compiler. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 341–345. ACM Press, 2002. ISBN 5-58113-473-8. doi: http://doi.acm.org/10.1145/563340.563473.

[3] Akim Demaille. Making compiler construction projects relevant to core curriculums. In *Proceedings of the Tenth Annual Conference on Innovation and Technology in Computer Science Education (ITICSE'05)*, pages 266–270, Universidade Nova de Lisboa, Monte da Pacarita, Portugal, June 2005. ISBN 1-59593-024-8.

[4] Akim Demaille and Roland Levillain. *The Tiger Compiler Project Assignment*. EPITA Research and Development Laboratory (LRDE), 14-16 rue Voltaire, FR-94270 Le Kremlin-Bicêtre, France, 2007. `http://www.lrde.epita.fr/~akim/ccmp/assignments.pdf`.

[5] Akim Demaille and Roland Levillain. *The Tiger Compiler Reference Manual*. EPITA Research and Development Laboratory (LRDE), 14-16 rue Voltaire, FR-94270 Le Kremlin-Bicêtre, France, 2007. `http://www.lrde.epita.fr/~akim/ccmp/tiger.pdf`.

[6] Akim Demaille, Roland Levillain, and Benoît Perrot. A set of tools to teach compiler construction. In *Proceedings of the Thirteenth Annual Conference on Innovation and Technology in Computer Science Education (ITICSE'08)*, Universidad Politécnica de Madrid, Spain, June 2008.

[7] Robert B. K. Dewar and Edmond Schonberg. Computer science education: Where are the software engineers of tomorrow? *CrossTalk, The Journal of Defense Software Engineering*, 21(1):28–30, January 2008. URL `http://www.stsc.hill.af.mil/CrossTalk/2008/01/0801DewarSchonberg.html`.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[9] LRDE. Tiger project home page, 2000. `http://tiger.lrde.epita.fr/`.

[10] Alexander Stepanov, Meng Lee, and David Musser. *The C++ Standard Template Library*. Prentice-Hall, 2000.

[11] Bjarne Stroustrup. Bjarne Stroustrup's FAQ. `http://www.research.att.com/~bs/bs_faq.html`, September 2008.

[12] The Boost Project. Boost C++ libraries. `http://www.boost.org/`, 2008.

[13] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient dynamic dispatch without virtual function tables: the SmallEiffel compiler. *ACM SIGPLAN Notices*, 32(10):125–141, 1997. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/263700.263728. URL `http://smarteiffel.loria.fr/papers/oopsla97.pdf`.