Making Compiler Construction Projects Relevant to Core Curriculums

Akim Demaille
EPITA Research and Development Laboratory (LRDE)
14/16, rue Voltaire
F-94276, Le Kremlin-Bicêtre, France
akim@epita.fr

ABSTRACT

Having 300 students a year implement a compiler is a debatable enterprise, since the industry will certainly not recruit them for this competence. Yet we made that decision five years ago, for reasons *not* related to compiler construction. We detail these motivations, the resulting compiler design, and how we manage the assignment. The project meets its goals, since the majority of former students invariably refer to it as *the* project that taught them the most.

Categories and Subject Descriptors

K.3.2 [**Computers & Education**]: Computer & Information Science Education—*Computer Science Education*; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms Design, Management

Keywords Compiler Design, Object Oriented Programming, Educational Projects, Design Patterns.

1. INTRODUCTION

EPITA is a private school teaching computer science to graduate students, with a strong practical emphasis via projects. Like most French engineering schools, it is composed of three phases: a two year preparation cycle with mostly theoretical basic courses, a year and a half core curriculum, and finally a year and a half specialization cycle.

Several years ago a reflection on the core curriculum projects highlighted the need for a long and challenging project, requiring material from virtually all the computer science courses. Compiler construction appeared as a miraculous solution to a complex system of goals/constraints. Five years later, this project is still alive, and quite successful.

Section 2 justifies having 300 students build a compiler. Section 3 describes the constraints this compiler should fulfill, and Section 4 presents parts of its original design that address them. The management of the project is sketched out in Section 5. Finally, Section 6 proposes guidelines, and Section 7 concludes.

2. YET ANOTHER COMPILER PROJECT?

Writing a compiler is a common assignment in computer science, dating back to the early days of compiler construction courses. Yet our project differs from most of its peers [2, 4, 6] in several regards, starting with the motivations: *compiler construction is a by-product*, not the primary goal. This project was introduced because it met a set of requirements that were not satisfied by the existing (shorter) projects:

Complete Project Parts of development that students tend to neglect should be emphasized: specifications, documentation and testing. A long project makes the latter two points critical: students feel the need for good (developer's) documentation (*why did we write/change this code 4 months ago?*), and extensive test suites (*why does stage* $n-2$ *fail* now*?*). None will be provided, they will have to develop theirs.

Several Iterations This is our only 6 (optionally up to 9) month project, with several deliveries and assessments. Therefore, students *feel* the constant need to fix errors found in earlier stages. Compared to throw-away assignments that they are not required to fix once rated, this completely changes their approach to programming by introducing software life cycles: dirty or undocumented code hinders the development of the next phase. They will have to take our comments into account.

Algorithmically Challenging Well known data structures (trees, graphs, automata, scoped symbol tables, etc.) and algorithms (LR parsing, traversals, sorting, term rewriting, graph coloring, etc.) must be used, to have students understand by themselves the importance and the relevance of the techniques they learned during the (somewhat theoretical) algorithmic courses.

Development Tools A wide spectrum of tools encompassing commonly used utilities (Automake, Doxygen, Flex, Bison, Debuggers, Valgrind, Version Control Systems, etc.). Other projects already use some of them, but nowhere else are so many of them required concurrently.

Team Management The assignment is addressed by groups of 4 over a 6 to 9 month period: human issues become critical. Experience shows that most students become much more receptive to "human management" courses afterward: sometimes we even have to intervene when the situation degenerates.

C++ EPITA selected C++ for its core curriculum for its expressive power and wide acceptance in industry, not for academic considerations about its qualities. It is not adequate for compiler construction *study*, but this seldom demanded competence is not a primary goal. For EPITA students, C++, Design Patterns, and OO Design are much more important — and highly demanded skills.

Object Oriented Design and Design Patterns The assignment must train students to OO design, and its most common idioms, the design patterns [10]. Use of UML is a plus.

English Every step of this project, except lectures and oral examinations, is done in English. This is a real obstacle for some students, but they have to overcome this difficulty since computer science requires some fluency in English.

Compiler construction is a miraculous answer to such a set of constraints. It also offers features of its own. Compilers (components) are everywhere under various disguise (XML processing, structured formats, interpreters etc.) [8]; many students are likely to have to work on them in their career. In addition, like assembly language notions, compiler study provides insight on computer functioning, kills some urban legends (e.g., tables better and faster than code); in short it helps programmers understand how to do a better job.

Hence, the project aims at the implementation of a compiler, *a secondary issue*, in C++, an essential goal. That compiler construction is not the main objective must be reiterated sometimes, either to recalcitrant students who criticize the pertinence of compiler construction on their resume, or to enthusiastic students who regret the absence of more advanced compiler material.

3. WHAT COMPILER?

There are numerous small languages and compiler designs created for compiler construction projects, probably nearly as many as there are teachers. We refrained from writing our own, a tremendous task in itself. Rather we looked for an existing well established project.

3.1 What Language?

The question was actually quickly answered, since we were enthusiastic about Andrew W. Appel's "Modern Compiler Implementation" book [3]:

- It is a recent text book addressing modern issues and modern techniques that more famous, but older, classic text books [1] do not mention.
- Its coverage is perfect for our students, who need to understand one good technique well, not all the known variations. For instance, since an implementation of scoped symbol tables amounts to nesting C++ maps in C++ stacks, for our purpose, most books are too comprehensive [1, 7].
- Its balance between theory and practice suits us, and in particular a full compiler implementation is detailed.
- Its 250 page presentation of a full compiler implementation does not frighten the students.
- It provides another 250 pages of more in-depth material for those interested.

Therefore we chose the language Andrew W. Appel designed for his book, Tiger, a small yet very complete subset of Pascal dressed in a clean ML-like syntax.

3.2 What Compiler Design?

The Tiger books [3] come in three implementation flavors: procedural with C, functional with ML, and object-oriented with Java. The C book samples are barely readable due to the low-level nature of the language. The ML edition is beautiful but inadequate to teach OO design. Unfortunately the Java book presents a very poor design; it is missing the unavoidable visitor design pattern, fails to use inheritance properly and so on. Of course, the book teaches compiler construction, not object-oriented design. Unfortunately we found no good compiler text book for C++; it seems that most authors refuse to use the standard library and redevelop low level components.

Because OO design is a more important objective than compiler construction (Section 2), we redesigned Appel's compiler, inspired by the original. In contrast to other approaches [6], we kept the same ambitions as Andrew W. Appel: the source language is fairly complete, and we target a genuine (MIPS) assembly language. In addition to meeting the aforementioned goals (Section 2), the compiler has to:

Never Repeat Itself Every stage must present/exercise something new. Because of time pressure, we cannot afford theme duplication. For instance we use several different memory management techniques (regular `new/delete`, reference counting, factories that reclaim their products, C++ `auto_ptr`).

Be Reasonably Dimensioned Some logistical details are time consuming and present little interest, in which case we provide the code. For instance 70% of the repetitive 3kloc AST is provided in full, leaving out only challenging code extracts. Because the project is decomposed by phases, we deliver the code (with gaps) incrementally: for each new stage, students receive patches to apply to their code.

Be Easy to Debug The compiler is similar to a long Unix pipe, with structured data flowing through different modules. Students need to "see" the data to control their job. This sole requirement represents about 10 command line options in their final compiler, and twice as many in our more feature-full reference compiler.

Be Easy to Check Automated tests for each phase are needed, which requires tools such as an intermediate code interpreter. When unavailable, we developed such tools as Free Software, and we also contributed to existing projects to provide more pedagogical features (Section 5.2).

Be Extremely Modular Because the code with gaps is provided stage after stage, everything is made to ease its integration into the students' code. This is especially constraining for the compiler driver[1] that currently supports about 60 options.

Follow Students in their C++ Learning Students receive their first C++ lectures when assigned the Tiger Project. Therefore stages should start with novice C++ material, and become increasingly advanced, ending with powerful C++ techniques (e.g., static programming, smart pointers, template template parameters, stream extensions, functional aspects, etc.).

The following section presents the resulting design.

[1]Roughly, the *driver* decodes command line options and invokes the components in the right order, satisfying prerequisites (such as "if `type-checking` is required, then `parsing` must be invoked beforehand").

4. THE DESIGN

The full presentation of our implementation goes beyond the scope of this paper. The interested reader is referred to our web site [9]. It is the result of a yearly redesign to address shortcomings of the implementation, or changes in pedagogical objectives. We also regularly change some aspects to discourage cheaters from using compilers from previous classes. Therefore there are many differences with Appel's design, two of them are detailed below because of their importance in education.

4.1 Semantic Analysis

Andrew W. Appel describes a single `semant` module for type checking and translation to the TREE intermediate language (Section 5.2.3). We first followed his design with a `semant_visitor`, but it was a disaster for students. Indeed, they were first given a partial `semant_visitor` adressing type checking, which they completed. Then we provided a patch to migrate the partial type checker into a partial type checker *and* translator. Since they had vastly edited the type checker, the patches no longer applied. To avoid this situation, we now only provide new modules and refrain from changing code from earlier stages: students might have customized it. Therefore the `semant_visitor` was split into (three) smaller components traversing the AST:

`bind_visitor` It binds identifier uses (e.g., a variable name) to their definition (e.g., variable declaration). This factors once and for all the handling of scopes. Before the introduction of this visitor, scopes were handled several times and students often had it right in one place, but wrong elsewhere.

`type_visitor` It checks the type consistency, and annotates the AST with types.

`translate_visitor` It translates the AST to TREE code using binding and type annotations.

Visibly more adequate for students, this combination is also a better design: in addition to factoring scopes processing, it eases extensions. For instance, function overloading support is implemented by subclassing the `bind_visitor` and `type_visitor`: the `translate_visitor` is identical.

4.2 The Task Manager

In our case not only is modularity a healthy principle, it is also required by the fact that we deliver code in stages. For instance, when type checking is assigned, students should be able to integrate easily the incomplete `type` module that we provide.

While modularity is claimed by many compilers, usually the driver is all but modular: it hard-codes calls to all the components. Our history demonstrated we cannot afford such a driver, because it is too hairy to maintain (imagine implementing "if displaying the AST is required, then activate parsing" for 60 options), and unsuitable for an incremental delivery (`type` module invocations cannot be left in the driver if that module is not present). It is also incompatible with the addition of optional modules.

To address these issues, modules have a two level interface. The *library level* is modeled after functional programming principles: there are only type definitions, pure functions (free of side effects), and no variable. On top of it is the imperative style *task level*, composed of global variables and actions changing these variables. When loaded, each task registers itself under a unique name, together with its prerequisites (much like Make "phony" targets, and LLVM's Pass Manager [12]). When the user passes command line options, the sequence of tasks to invoke is completed and executed.

5. LOGISTICS

Managing such a project, spanning 6 to 9 months, with 300 students per class, i.e., from 7 to 10 deliveries of about 80 compilers is a challenge in itself. Everything that can be automated must be automated, everything that should be known must be written, everything that can help must be done.

5.1 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) 18 students of the previous class who particularly enjoyed the project and are willing to assist younger students. They are also in charge of oral examinations.

EPITA being strongly oriented towards practical applications, there are numerous projects of various sizes that all need coordination between the team and the students. We use web sites to publish information, mailing lists to coordinate ourselves, and newsgroups for asynchronous discussion with the students. The assistants also developed a complete web site to manage the class, including automated examinations.

The Tiger project starts in January, after students have completed a "Languages, Automata, Grammars, Parsing" course. To learn the language they first have to write a significant program in Tiger, e.g., an infix calculator, Huffman coding, or LZW compression. They will use this assignment, together with other provided big programs, to exercise their compilers. Contrary to tiny languages such as J-- [5], Tiger is rich enough so that compiling higher level languages *to* Tiger is not needed. Almost concurrently they start writing the scanner and parser. At the same period C++ lectures start. The mandatory part (down to intermediate representation, see section 5.2.2) ends in June and the optional MIPS back-end in September. Examples of assignments include writing the `bind_visitor` from scratch, most of the type checking, smart pointers, additional containers, and the last (optional) stage is the full implementation of a graph-coloring register allocator. In July, the class is invited to debate about the project. Changes are always suggested. From October to January, the team hacks the compiler for the following year, taking comments into account.

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. Of course, the project newsgroup is very active a couple of days before the delivery. On the Friday at noon, students must have uploaded their package on the web site. It launches the automated testing procedure, which evaluates each stage of the compiler. It computes a global correction grade from these figures, 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. Each assistant spends about 45 minutes with the group, asking questions, evaluating answers, browsing the code, and helping students to understand and overcome their problems. Finally the *product* of the compiler correction grade and examination grade gives the final grade for one stage. There has been much debate about this formula, but EPITA emphasizes the importance of producing effective software: if you fail half the test, you can't decently expect to have half the highest grade.

A 170+ page document details all the assignments and makes explicit what students are expected to learn. It also proposes ideas of optional extensions. Because the project is created for the whole class, *suggesting options is mandatory.* Indeed, the compiler has to remain very simple, serving other purposes than compiler construction (Section 2). But this frustrates the most enthusiastic students who do not find the challenge (and pride!) they expected. Modularity, especially thanks to the Task manager, lets students plug-in their extensions (Section 4.2). Most notable extensions students have designed and implemented include copy propagation, support for an `import` feature, object orientation, function overloading, static link optimization, tail recursion elimination, bounds checking, improvements to the "stock" register allocation algorithm, various forms of pretty printing etc. As extreme examples, a student implemented a Tiger front end for GCC, another rewrote his compiler in C# to learn it, and Francis Maes published his astonishing results in compiling towards static C++ [13].

5.2 Tools

In such a complex environment (300 students), automation is salvation. Many auxiliary tools were written 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.

Most of these tools are private. For instance, students write the AST because it is an excellent means to understand inheritance, but we generate ours, to easily change parts of the compiler (and because we understand inheritance). There are also tools that we want them to use to save time, such as Valgrind or Automake[2]. Some of these tools had to be developed (a generator of code generators in C++ is ongoing work) or extended. A more in depth presentation of these tools and their relevance to computer science education is deferred to another paper, but we present three significant examples below.

5.2.1 GNU Bison

No tool was available to produce LALR(1) automaton diagrams for Yacc grammars, so we extended GNU Bison. We also enriched its textual automaton description with missing information such as the lookahead sets, and conflict resolutions. We transformed GNU Bison to support several output formats and implemented a C++ parser output. To have students make precise error messages, we implemented location tracking. Finally, measuring memory leaks in the student compilers, we found that Bison parsers leak memory during error recovery, a well known issue going back to the original Yacc. We introduced the `%destructor` directive to fix this problem.

[2]The first year they wrapped their packages by hand: the archives used to contain incredible material, from core dump and other object files, to packages of other groups!

5.2.2 Havm

Andrew W. Appel designed TREE, a simple register-based high level intermediate code. In a later stage, the compiler transforms TREE code into a lower level subset of TREE. Havm, by Robert Anisko, is an interpreter for these high and medium level intermediate languages. It features a run-time library, support for recursion, a debugging mode, and a simple code performance measurement.

5.2.3 Nolimips

Nolimips is a basic MIPS architecture simulator written by Benoît Perrot composed of a MIPS assembler and of a virtual machine. Compared to SPIM [11] it features uninitialized memory access and calling convention respect checks (a la Valgrind), unlimited pseudo register number (`$x42`, `$x51`, etc.), and variable (real) register number. The last two features deserve more attention since they prompted the development of Nolimips[3].

The compiler first produces assembly code with unlimited pseudo-registers, then performs "register allocation": allocating real registers to these pseudo-registers. To help students and examiners checking the compilers we needed an interpreter that would allow for unlimited registers.

The possibility to limit the number of MIPS registers is motivated by two reasons. (i) Our register allocation is based on graph coloring, and such a graph, even for simple functions, is large and "unreadable". This is due to the 8 callee-save registers saved in 8 pseudo-registers (8 nodes) colliding with (connected to) every other pseudo. Controlling the number of hard registers makes the process much more tractable for humans, and for students. (ii) The MIPS architecture features 32 registers and the whole set is seldom needed, so register allocators are unlikely to be exercised on extreme situations. Bounding the number of registers addresses both concerns. None of these reasons require a register starving MIPS, what matters is that the *compiler* produces code for such a register starving MIPS. But to *check* that the compiler respects this limitation requires a limited MIPS.

6. DISCUSSION

The Tiger project meets its goals: (most) students have good C++ programming skills and know the principal design patterns, do not hesitate to implement complex algorithms or to use an auxiliary tool, pay attention to documentation and tests, and understand how compilers work. This is the result of hard work, including regular overhauls, to take into account student criticism. Even though the first class suffered from many defects (we were one week ahead of them!), the majority of them invariably refer to Tiger as *the* project that taught them the most.

The constant redesign also improves the understandability, as can attest the assistants who followed the evolution. Because the phases also change and incorporate new pedagogical material, grades are incomparable between classes.

We drew some guidelines from our experience that we now follow. They can serve groups managing long-term projects with objectives similar to ours (Section 2):

[3]For a number of reasons, technical and political (it is not Free Software) we did not extend SPIM, but rather developed another emulator.

Define the Objectives Make sure students know the objectives of the project, otherwise they might focus on auxiliary points, and miss the important ones.

Use Hierarchized Media Web sites should contain often needed documents, including an FAQ. Any question asked twice should be answered there. Then use newsgroups or public mailing lists for questions to delegate the effort: students will often answer, otherwise assistants will. Make clear that private messages are for private matters exclusively.

Involve Other Students Students who enjoyed the project are likely to spend a considerable amount of time to assist younger students. Sometimes they also have a better understanding of their problems. Working with them is enriching. Of course, it doesn't work the first year, but bootstrapping is always a problem in compiler construction.

Be Modular Incremental code delivery requires a fixed base code. A task-driven top level is a fine answer (Section 4.2).

Keep it Simple Do not aim too high: a long and hard project excludes students. Do not leave nice and original extensions, or even auxiliary code, to students. Such pieces of code can distract or incredibly hinder weak students who can't even tell the difference between the core of the project and decorations.

Promote Extensions Conversely, meet the expectations of enthusiasts by proposing extensions, possibly ambitious ones. In addition, they are happy to contribute their code to the reference compiler to ease the checking of such extensions in the future.

Automate Nothing is overkill: everything that can be automated must be. That may require additional development, but the time spent to assist students will be overwhelming anyway. In the long run, tools are always a better option.

Constantly Refactor The maintenance of such a project is extremely demanding. Be ready to adjust the code to follow pedagogical changes, to address issues raised during the year, etc.

Write Your History It is often too late that one realizes data from early experiments is lost.

7. CONCLUSIONS

We explained why a compiler construction project is relevant to a core curriculum, even if compiler study is not the starting point. We emphasized a few key aspects of the design such a long project should feature, and their impact on student involvement. We also proposed simple guidelines for similar projects.

8. ACKNOWLEDGMENTS

Listing all the people beyond the core team who participated in the Tiger project is impossible. First Thierry Géraud taught me C++ and OO design. He designed and implemented the AST and the first visitors; he also had the idea of a task manager. Benoît Perrot did an enormous job maintenance and documentation. Many other people contributed code or ideas: R. Anisko, C. Bail, A. Brouard, A.-L. Brouhrant, S. Broussaud, N. Burrus, C. Calméjane, V. David, A. Dumont, A. Duret-Lutz, A. Fabre, W. Fink, D. Gazard, P. Guedon, F. Hesling, T. Lanfrey, R. Levillain, F. Maes, S. Molina, Q. Peyrot, S. Plessis-Fraissard, Y. Popo, R. Poss, N. Pouillard, Y. Régis-Gianas, J. Roussel, P.-Y. Strub, M. Tessari, C. Vasseur, and G. Walbrou. And thanks to S. Frank for his proofreading.

9. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] A. Aiken. Cool: A portable project for teaching compiler construction. In *ACM Sigplan Notices*, volume 31, pages 19–26, July 1996.

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

[4] J. Aycock. MBL: A language for teaching compiler construction. Technical Report 1995-574-26, Department of Computer Science, University of Calgary, 1995.

[5] J. Aycock. The ART of compiler construction projects. In *ACM SIGPLAN Notices*, volume 38, pages 28–32, Dec 2003.

[6] D. Baldwin. A compiler for teaching about compilers. In *SIGCSE'03*, pages 19–23, Reno, Nevada, USA, February 2003.

[7] K. D. Cooper and L. Torczon. *Engineering a Compiler.* Morgan Kaufmann, 2004.

[8] S. 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.

[9] A. Demaille. The Tiger compiler project home page. `http://tiger.lrde.epita.fr`.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[11] J. R. Larus. SPIM S20: A MIPS R2000 simulator. Technical report, Computer Sciences Department, University of Wisconsin–Madison, 1990.

[12] C. Lattner and V. Adve. Architecture for a Next-Generation GCC. In *Proc. First Annual GCC Developers' Summit*, Ottawa, Canada, May 2003.

[13] F. Maes. Program templates: Expression templates applied to program evaluation. In J. Striegnitz and K. Davis, editors, *Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages (DP-COOL; in conjunction with PLI)*, number FZJ-ZAM-IB-2003-10 in John von Neumann Institute for Computing (NIC), Uppsala, Sweden, August 2003.