A Compiler Project with Learning Progression

Derek Rayside

Electrical and Computer Engineering

University of Waterloo

drayside@uwaterloo.ca

Abstract—We describe the design of an undergraduate compilers course for computer engineering students in which the project is to write a simulator and synthesizer for (a subset of) VHDL. The traditional project for such a course is to write a compiler for (a subset of) some procedural programming language. The choice of source language is superficially intended to drive student engagement for computer engineers.

The main pedagogical advantage of this project is that it embodies a *learning progression*: repetition with increasing complexity. This project involves two additional languages: a regular language for boolean waveforms (used for circuit simulator input and output), and a context-free language for boolean formulas. Parsing and transformations are performed on these simpler languages before attempting them on the subset of VHDL.

At the end of the project the students can simulate and synthesize simple circuits such as a ripple-carry adder or a multiplexer.

I. INTRODUCTION

There are a variety of undergraduate compiler lab projects. Almost all of them involve compiling a subset of some imperative programming language to assembly. This project is different. On a superficial level, the difference is that these labs use a subset of VHDL as the input language, and produce circuit gate diagrams and simulations as outputs.

The deeper pedagogical difference is that this project is designed around a *learning progression*: repetition with increasing complexity, which hopefully leads to mastery. This instructor acquired the idea of a learning progression from a hockey coach several decades ago. For example, suppose that the goal of the practice is to have the players skate quickly around the circles with the puck. To progress to that goal the team might start with skating around circles, then skating in a straight line with the puck, then skating around the circles slowly with the puck, and finally skating around the circles quickly with the puck. The skills are first practiced in isolation and at slower speeds, and then finally all together at high speed. The idea of learning progression also seems quite popular in educational circles in recent years (*e.g.*, [1]).

This project actually comprises two parallel learning progressions: compiler concepts and advanced object-oriented programming skills. The prerequisite courses cover basic programming skills, but do not teach the advanced programming skills necessary to write a compiler. Most compiler projects, text books, and courses, assume (perhaps wrongly) that the students already have the required programming skills. (Appel's text book [2], which we use, is a notable exception.)

Compiler projects are usually structured around the logical organization of a compiler, proceeding from the front (parsing)

to the back (code generation) in a single pass. Concepts are not revisited. Programming skills are not taught. While this approach is constructivist [3], it does not have a learning progression.

The key technical decision that enables these learning progressions is to restrict our subset of VHDL ($\mathcal{V}$) to simple combinational circuits. A simulator for such circuits reads and writes strings of bit values for its input and output pins. We design a simple regular language, $\mathcal{W}$, for recording these waveforms (bit strings). Combinational circuits can be described by boolean formulas, and we use the assignment statement subset of our $\mathcal{V}$ grammar to describe these formulas. We name this language $\mathcal{F}$. $\mathcal{F}$ is a simple context-free language, grammatically similar to the toy arithmetic calculator language often used in compiler text books. Students are more motivated about working with these simple languages because of their role in building VHDL tools.

Most of the labs actually deal with $\mathcal{W}$ and $\mathcal{F}$: only the last three labs work with $\mathcal{V}$ directly. First we parse $\mathcal{W}$, then we parse $\mathcal{F}$, then we parse $\mathcal{V}$. First we transform $\mathcal{W}$, then we transform $\mathcal{F}$, then we transform $\mathcal{V}$. The progression of theoretical complexity corresponds to a progression of practical complexity. These languages are summarized in Figure 1. An end-to-end example of a full adder is given in Figures 4–8.

Fig. 1 Summary of languages processed in project

Language	Complexity	Example
$\mathcal{W}$ boolean waveforms	regular	$X: 0\ 1\ 0\ 1;$
$\mathcal{F}$ boolean formulas	context-free	$X \leq p \text{ or } q;$
$\mathcal{V}$ hardware description	context-free	<i>see Figure 4</i>

II. CURRICULUM CONTEXT

The University of Waterloo has three undergraduate compilers courses: **ECE351** (this course), CS241, and CS444. CS462 is an additional theory course on formal languages and parsing. Both CS and ECE offer graduate level courses on compilers and related topics.

ECE351 fills roughly the same slot as CS241: an introductory compilers course. CS241 and **ECE351** might be considered anti-requisites of each other (although the calendar does not list them as such at the moment, because there is no practical need for this constraint). Both courses are adequate preparation for CS444.

ECE351 is a mandatory course for the computer engineering students. Usually a handful of electrical engineering students take the course as an elective. The software engineering and computer science students take CS241 instead.

Part of being a mandatory course is that **ECE351** gets all of the computer engineering students, not just the ones who have strong interest and ability in software. These are students who would have enrolled in software engineering or computer science if their primary interest was software. Therefore, there is a greater need to teach the requisite programming skills than with an elective course.

A strategic difference between **ECE351** and CS241 is that **ECE351** does not include assembly language in the core material, because the ECE students study assembly in ECE222, whereas CS241 is the main exposure to assembly for the students who take that course. While the top students are similar in both programs, the bottom end of programming ability in the ECE students is weaker. These labs spend more time building programming skills at the expense of excluding assembly language from the core material. This summer term we will have a bonus lab that targets assembly language rather than Java for the $\mathcal{V}$ simulator for students who have the interest and ability.

The computer engineering students at the University of Waterloo use VHDL more than any other language. It is used in ECE124, ECE224, and ECE327. ECE327 is about designing digital systems and is taught in VHDL. Almost all of the students in **ECE351** are concurrently enrolled in ECE327, which is also required for computer engineering students. The selection of VHDL for this course is intended to drive student engagement, since these are hardware-oriented students. In ECE327 they use professional-grade VHDL tools, and in **ECE351** they build toy-grade VHDL tools.

The computer engineering students will have had four courses with software programming before enrolling in **ECE351**: two introductory programming courses (ECE150 & ECE155), as well as a data structures and algorithms course (ECE250), and an operating systems course that involves programming (ECE254). Each of these courses is taught in one of C, C++, C#, or Java. So the students have been exposed to a variety of software programming languages before **ECE351**. Nevertheless, there are still advanced programming concepts that are required for writing compilers that they have not been previously taught. These labs are designed to provide a learning progression for those programming concepts as well as for the compiler concepts.

The University of Waterloo engineering program runs on a cohort system. Each cohort takes almost all of the same classes at the same time with the same instructor. So the students coming into **ECE351** have had a very uniform instructional background. As has been observed many times in other educational and industrial settings, the range of software ability varies widely.

III. PROJECT DESCRIPTION

The overall structure of the project is depicted in Figure 2. Nodes represent file types. All of these file types, with the exception of .class and PNG files, are text files. Edges represent translators between different file types. Labels on edges describe the translation(s) performed. Solid edges represent translators that students implement in this project. Dotted edges represent translators provided by third-parties such as Sun/Oracle (javac) or AT&T Research (Graphviz/dot). The

Fig. 2 Overview of **ECE351** Labs.

three-part edge between .class and $\mathcal{W}$ nodes is intended to indicate that the .class file we generate will read a waveform ($\mathcal{W}$) file as input and write a waveform ($\mathcal{W}$) file as output. An end-to-end example of a full adder is given in Figures 4–8.

The general direction of work in the project is from the bottom of the figure towards the top. In other words, we start with the simplest language ($\mathcal{W}$) and work towards the most complicated language ($\mathcal{V}$).

A. Labs

The project is divided into 11 individual labs, which are described in Figure 9. Parsing is done in labs 1, 3, 5, 6, and 9. Transformations or translations are done in labs 2, 4, 7, 8, 10, and 11.

The learning progression for parsing starts with writing a regular expression to recognize $\mathcal{W}$ programs (lab 1) and proceeds to using a parser generator for $\mathcal{V}$ programs (lab 9). Each parsing lab involves first writing a recognizer and then adding actions to the recognizer to make a parser. The first parsing labs have the students write recursive descent parsers by hand for $\mathcal{W}$ (lab 1) and $\mathcal{F}$ (lab 3).

The use of a parser generator is not introduced until lab 5, where the students use Parboiled to parse $\mathcal{W}$. By this time they are quite familiar with $\mathcal{W}$ and with the skeleton code: they have written a regular expression recognizer for $\mathcal{W}$, a recursive descent recognizer and parser for $\mathcal{W}$, as well as transformed $\mathcal{W}$ to SVG. The only thing new here is the parser generator. In labs 5, 6, and 9, the students use the parser generator for languages $\mathcal{W}$, $\mathcal{F}$, and $\mathcal{V}$, respectively.

The learning progression for transformations proceeds from translating $\mathcal{W}$ to SVG to performing function inlining (‘elaboration’) on $\mathcal{V}$. The AST for $\mathcal{W}$ comprises just two classes with no inheritance. The AST for $\mathcal{F}$ comprises 21 classes and has a deep inheritance structure for the various types of expressions. The AST for $\mathcal{V}$ comprises 28 classes, 20 of which are shared in common with $\mathcal{F}$.

The learning progression for programming skills follows the increasing complexity of the grammars and the transformations. The design patterns required for each lab are identified in Figure 9, and are noted as either *explicit* or *latent*. The ‘explicit’ label indicates that the students need to know the name of the design pattern and understand it in order to complete the lab. The ‘latent’ label indicates that the lab uses that design pattern, but that the students do not need to know its name or definition in order to complete the lab. For example, lab 4 makes use of the interpreter design pattern, but it is latent. We do not actually discuss the interpreter design pattern until lab 7 when the visitor design pattern is introduced. At that time we discuss the trade-offs between the two patterns.

B. Parser Generator

We are using the Parboiled [4] parser generator for two reasons. First, it uses an internal DSL (domain-specific language) for describing grammars. Most parser generators use an external DSL for describing grammars. The term ‘internal’ here is with reference to the host language, which in this case is Java. In other words, Parboiled is a Java library. Most other parser generators require the students to learn a new grammar specification language, and put an extra step in the build process to generate (Java) code from that grammar specification.

Second, Parboiled works very much like a push-down automata: the only storage mechanism is the stack; at the end of the parse the result is the object on the top of the stack.

C. Extensions and Alternatives

The basic labs as described in Figure 9 do not involve generating assembly code. Some of the students have said that they miss that. An obvious extra lab is to translate $\mathcal{F}$ to assembly rather than to Java. We will offer this as a bonus lab in the next offering of **ECE351**. That lab would incorporate instruction selection and register allocation, which at present are ideas we discuss only on paper.

Other topics that we discuss only on paper, which further labs could be designed around, include dataflow analysis, symbol tables, type checking, and instruction scheduling. We are considering adding some of these labs towards the end of the project, and then dividing the students into groups to tackle them.

IV. STUDENT WORKLOAD

ECE351 is a 12 week course that nominally requires a 10-hour per week commitment from the students, which includes lecture, tutorial, lab, and study time. There are three hours of lecture per week. We have structured the labs into weekly exercises targeting five hours each. Although there are 12

Fig. 3 The median (dark bar) student time on each lab was close to the five hour target in winter 2013, except for labs 0, 4 and 9: prelab exercise, simplifying boolean formulas by term rewriting, and parsing $\mathcal{V}$, respectively. Boxes capture the middle 50%. Circles indicate outliers.

weeks of class we give only 11 labs so that they get a week off to study for midterm tests.

Figure 3 depicts the hours worked on each lab by the students of the winter 2013 offering of **ECE351**. There were sixty students enrolled in the course. Reporting of hours was mandatory, although enforcement was lax.

The data in Figure 3 show that half the students are completing the labs within the allotted five hours, and that 75% of the students complete their work within eight hours. The two exceptions to this general rule are labs 4 & 9. Lab 4 is optimizing $\mathcal{F}$ by term rewriting, and lab 9 is parsing $\mathcal{V}$.

The slowest students sometimes report extraordinary values of up to forty hours for some labs. We are certainly concerned about such values, but suspect that these kinds of values also occur in other mandatory software related courses.

Lab 4 is the first time that the students must do advanced object-oriented programming. We suspect that part of the conceptual difficulty here is understanding the self pointer as polymorphic, which is the core of the template design pattern.

Anecdotal feedback from students suggests that one of the reasons lab 9 took longer than expected is that the tests for lab 9 revealed bugs in their implementations of previous labs, and so a certain amount of the time reported as lab 9 should really have been allocated to previous labs. We are working on enhancing the tests for all labs to mitigate this situation in future. We are also going to try to have the students develop greater intellectual discipline in their approach to lab 9: specifically, that they sketch out on paper a plan for how they are going to manage the stack before they start programming.

The project as a whole comprises about 8400 lines of Java code, including test harnesses (as reported by David A.

Wheeler’s SLOCCount tool). We give about 7000 of these lines to the students as skeleton, leaving them an average of about 125 lines to write per lab (week) — just under 2.5 minutes per line.

V. VHDL SUBSET

Our language $\mathcal{V}$ is a very small subset of VHDL: essentially limited to combinational circuits. The two most complicated $\mathcal{V}$ input files the students are expected to handle are a four-bit ripple-carry adder and a multiplexer.

We started out by downloading an open source ANTLR grammar for VHDL and quickly found out that was far too complicated for the purposes of this project — moreover, it was too complicated to even simplify. So we started building a simple grammar from the bottom up. Four consecutive groups of students worked on developing this grammar, each group making further simplifications along the way. The first group (volunteer) used ANTLR [5]. The second group (volunteer) translated their ANTLR grammar to TXL [6]. The third group translated this TXL grammar to Parboiled (enrolled), and the final group (TA) simplified the Parboiled grammar to focus solely on the needs of this project. Subsequent to these four rounds of revision and simplification we have not had a need to modify the grammar further.

The grammar for $\mathcal{V}$ is listed in Figure 10. Restrictions of $\mathcal{V}$ include:

- no combinational loops
- only bit (boolean) variables
- no stdlogic
- no arrays
- no nested ifs
- no aggregate assignment
- no generate loops
- no case
- no when (switch)
- no wait
- no timing
- no postponed
- identifiers are case sensitive
- one architecture per entity, and that single architecture must occur immediately after the entity
- inside process: either multiple assignment statements or multiple if statements; inside an if there can be multiple assignment statements

Our language $\mathcal{F}$ is a subset of this $\mathcal{V}$ grammar: just the assignment statements and expressions. So when the students get to implementing the $\mathcal{V}$ parser they can reuse the parser and AST classes for the $\mathcal{F}$ subset.

VI. CONCLUSIONS

Compiler projects are usually structured around the logical organization of a compiler, proceeding from the front (parsing) to the back (code generation) in a single pass. Concepts are not revisited. Moreover, most compiler projects and textbooks do not explicitly address the required programming skills (Appel’s text book [2], which we use, is a notable exception). While this approach is constructivist [3], it does not have a learning progression.

This project, by contrast, is explicitly structured around *learning progression*. Students perform parsing, transformation, and translation on three languages of increasing complexity: $\mathcal{W}$, $\mathcal{F}$, and $\mathcal{V}$. $\mathcal{W}$ is a regular language used to describe boolean waveforms. $\mathcal{F}$ is a simple context-free language used to describe boolean functions. $\mathcal{V}$ is a simplified context-free subset of VHDL that includes $\mathcal{F}$ as its expression language. Repetition with increasing complexity (hopefully) leads to mastery. This project comprises two parallel learning progressions: compiler concepts and advanced object-oriented programming.

It is difficult to empirically assess whether explicitly incorporating a learning progression leads to better educational outcomes in this context. While hundreds (if not thousands) of students have taken this course before this project was developed, we think that comparing the lab grades, exam grades, or course critiques against these historical offerings would not yield reliable insight. The empirical data that we do have suggests that most students are able to complete these labs within a reasonable approximation of the allotted time budget.

In pedagogical theory a learning progression is accompanied by formative assessments at each step of the way [1]. At present our assessments include tests for each of the labs. Our experience shows that these tests need to be enhanced, since students find creative ways to pass them while still having incorrect solutions. We are looking at adding some mechanically generated tests from tools such as Randoop [7] and Korat [8]. We are also looking to add more pencil and paper exercises and questions to accompany each lab.

Simply collecting the student hours on each lab has given us insight into the steps students have difficulty with, and in that sense at least the learning progression is successful. For example, in the summer of 2012 offering we discovered that the students’ knowledge of object-oriented programming was less advanced than required to write a compiler by the amount of time they spent on the early labs. The labs presented in Figure 9 have been reorganized to introduce these programming concepts more gently.

ACKNOWLEDGEMENTS

Prof Paul Ward suggested making labs based on VHDL to foster student engagement. Prof Mark Aagaard (ECE327 instructor) assisted with VHDL. TA Wallace Wu developed the VHDL-specific labs. Students Zhengfang (Alex) Duanmu, Lingyun (Luke) Li, Jiantong (David) Gao, Rui (Ray) Kong, Aman Muthrej, and Atulan Zaman helped develop the labs before they were used in a large class. Lab Instructor Jon Eyolfson and TAs Vajih Montaghani and Alireza Mortezi have been very helpful. Prof Patrick Lam has been generous with his time and his notes. Philip Guo provided helpful feedback on an earlier draft.

REFERENCES

- [1] T. Corcoran, F. A. Mosher, and A. Rogat, “Learning progressions in science: An evidence-based approach to reform,” Consortium for Policy Research in Education, Center on Continuous Instructional Improvement, Teachers College, Columbia University, Tech. Rep., May 2009, 86 pages. [Online]. Available: http://www.cpre.org/images/stories/cpre_pdfs/lp_science_rr63.pdf

- [2] A. W. Appel, *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.
- [3] J. Piaget, *The Origins of Intelligence in Children*. New York: International University Press, 1952.
- [4] M. Doenitz, "Parboiled parser generator," 2009. [Online]. Available: parboiled.org
- [5] T. Parr, *The Definitive ANTLR Reference*. The Pragmatic Bookshelf, May 2007.
- [6] J. R. Cordy, I. H. Carmichael, and R. Halliday, "The TXL Programming Language v10.5," Queen's University, Kingston, ON, Tech. Rep., 2007.
- [7] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th ACM/IEEE International Conference on Software Engineering (ICSE)*, W. Emmerich and G. Rothmel, Eds., Minneapolis, MN, 2007.
- [8] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov, "Parallel test generation and execution with Korat," in *Proceedings of the 6th European Software Engineering Conference (ESEC) and ACM/SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, I. Crnkovic and A. Bertolino, Eds., Dubrovnik, Croatia, Sep. 2007, pp. 135–144.

Fig. 4 Source code for full adder circuit ($\mathcal{V}$)

```

entity full_adder is port (
    A, B, Cin: in bit;
    S, Cout: out bit
);
end full_adder;

architecture full_adder_arch of full_adder is
begin
    S <= (A xor B) xor Cin;
    Cout <= ((A xor B) and Cin) or (A and B);
end full_adder_arch;

```

Fig. 5 Input waveform for full adder ($\mathcal{W}$)

```

A: 0 1 0 1 0 1 0 1;
B: 0 0 1 1 0 0 1 1;
Cin: 0 0 0 0 1 1 1 1;

```

Fig. 6 Boolean formulas for full adder ($\mathcal{F}$ generated from source code in Figure 4)

```

S <= ((not (((not A) and B) or ((not B) and A))) and Cin) or
      (((not A) and B) or ((not B) and A)) and (not Cin);
Cout <= (((not A) and B) or ((not B) and A)) and Cin
        or (A and B);

```

Fig. 7 Gates for full adder (generated from formulas in Figure 6)

Fig. 8 Input and output waveforms for full adder (generated from formulas in Figure 6 and input waveform in Figure 5)

Fig. 9 Descriptions of individual labs with the compiler and programming concepts introduced in each

#	Description	Compiler Concepts Introduced	Programming Concepts Introduced
0	Prelab		IDE, version control, merge conflicts, test harnesses, debugger
1	Parsing $\mathcal{W}$ by recursive descent	regular languages, regular expressions, EBNF, recognition, parsing, recursive descent, lexing, pretty-printing, AST	classes, representation invariants, object equality, immutability
2	Translating $\mathcal{W}$ to SVG (visualization)	trees, translation, iterator DP (explicit)	type tests, casting, XML, SVG
3	Parsing $\mathcal{F}$ by recursive descent	context-free grammars, LL(1) grammars, parse tree, isomorphism, associativity, commutativity, precedence, predict sets	inheritance, polymorphism, dynamic dispatch, singleton DP (latent), composite DP (latent), recursive structures, recursive functions, higher-order functions
4	Simplifying $\mathcal{F}$ programs (optimization)	tree transformation, intermediate forms, term rewriting, confluence, termination, convergence, identity element, absorbing element,	interpreter DP (latent), template DP (explicit)
5	Parsing $\mathcal{W}$ with a parser generator	parser generators	domain-specific languages, debugging generated code
6	Parsing $\mathcal{F}$ with a parser generator		
7	Translating $\mathcal{F}$ to Graphviz (technology mapping)	common subexpression elimination	visitor DP (explicit), tree traversals, hash structures, iteration order, object identity, non-determinism
8	Translating $\mathcal{F}$ to Java (circuit simulation)	program generation	preorder traversal
9	Parsing VHDL with a parser generator	desugaring	
10	VHDL elaboration	function inlining	
11	VHDL process splitting and translation to $\mathcal{F}$ (combinational synthesis)		
B	Translating $\mathcal{F}$ to assembly	instruction selection, register allocation	assembly, linking

* ‘DP’ stands for ‘design pattern’; ‘B’ stands for bonus lab

Fig. 10 Grammar for our VHDL subset. Keywords are in bold.

<i>Program</i>	→	<i>(DesignUnit)*</i>
<i>DesignUnit</i>	→	<i>EntityDecl ArchBody</i>
<i>EntityDecl</i>	→	‘entity’ <i>Id</i> ‘is’ ‘port’ <i>(‘(IdList ‘:’ ‘in’ ‘bit’ ‘;’ IdList ‘:’ ‘out’ ‘bit’ ‘)’ ‘;’ ‘end’ (‘entity’ Id) ‘;’</i>
<i>IdList</i>	→	<i>Id (‘,’ Id)*</i>
<i>ArchBody</i>	→	‘architecture’ <i>Id</i> ‘of’ <i>Id</i> ‘is’ <i>(‘signal’ IdList ‘:’ ‘bit’ ‘;’)?</i> ‘begin’ <i>(ComplInst)* (ProcessStmts SigAssnStmts)</i> ‘end’ <i>Id</i> ‘;’
<i>ProcessStmts</i>	→	<i>(ProcessStmt)⁺</i>
<i>ProcessStmt</i>	→	‘process’ <i>(‘(IdList ‘)’ ‘begin’ (IfElseStmts SigAssnStmts) ‘end’ ‘process’ ‘;’</i>
<i>IfElseStmts</i>	→	<i>(IfElseStmt)⁺</i>
<i>IfElseStmt</i>	→	‘if’ <i>Expr</i> ‘then’ <i>SigAssnStmts</i> ‘else’ <i>SigAssnStmts</i> ‘end’ ‘if’ <i>(Id)?</i> ‘;’
<i>ComplInst</i>	→	<i>Id</i> ‘:’ ‘entity’ ‘work.’ <i>Id</i> ‘port’ ‘map’ <i>(‘(IdList ‘)’ ‘;’</i>
<i>SigAssnStmts</i>	→	<i>(SigAssnStmt)⁺</i>
<i>SigAssnStmt</i>	→	<i>Id</i> ‘<=’ <i>Expr</i> ‘;’
<i>Expr</i>	→	<i>Relation (LogicOp Relation)*</i>
<i>Relation</i>	→	<i>Factor (RelOp Factor)?</i>
<i>Factor</i>	→	<i>(‘not’)? Literal</i>
<i>Literal</i>	→	<i>Id “ Constant ” ‘(Expr ‘)’</i>
<i>LogicOp</i>	→	‘and’ ‘or’ ‘xor’ ‘nand’ ‘nor’ ‘xnor’
<i>RelOp</i>	→	‘=’
<i>Constant</i>	→	‘0’ ‘1’
<i>Id</i>	→	<i>Char (Char Digit ‘_’)*</i>
<i>Char</i>	→	<i>[A–Z a–z]</i>
<i>Digit</i>	→	<i>[0–9]</i>