Course Experience Report: Full-class Compiler Collaboration

Yousef Alhessi
University of California San Diego
California, USA
yalhessi@eng.ucsd.edu

Joe Gibbs Politz
University of California San Diego
California, USA
jpolitz@eng.ucsd.edu

Abstract

Compilers are large software systems. In course projects it is often a challenge for students to build a significant compiler on their own with features like memory management, closures, inheritance, and more. We report on our experience splitting a relatively large compiler, with several of these advanced features, among project groups in a graduate compilers course. In addition to allowing students to engage with a larger system than groups would have been able to build on their own, we also believe based on anecdotal feedback that this had positive effects on student morale and community. There were several concrete logistics and content decisions we made that were effective, along with other recommendations and refinements for when we run the course again.

CCS Concepts: • **Software and its engineering** → **Programming teams**; **Compilers**; • **Applied computing** → **Collaborative learning**.

Keywords: compilers, computer science education, collaborative software development, webassembly, typescript

ACM Reference Format:
Yousef Alhessi and Joe Gibbs Politz. 2021. Course Experience Report: Full-class Compiler Collaboration. In *Proceedings of the 2021 ACM SIGPLAN International SPLASH-E Symposium (SPLASH-E '21), October 20, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3484272.3484961

1 Course and Context

We report on an offering of a 10-week graduate compilers course at a large public research university in the US.[1] Broadly, the course covers compilers topics like optimization, code generation, interactive evaluation, and runtime systems. Notably, it is a "backend" compilers course not focused on lexing or parsing. The course recommends, but does not require, undergraduate-level compilers background. The primary new aspect we report on is a large, class-wide project collaborating on a single compiler. This is similar to efforts in software engineering courses [2, 11]; to our knowledge this is the first published report on a project like this specifically in a compilers course.

In the offering of the course we report on, 57 students enrolled and 48 completed it. [2] The lead instructor had not taught the course before, while the graduate co-instructor had, but in a different style than we report on here. We mention this to highlight that this was the first offering of this particular design.

1.1 Course Structure and Schedule

The 10 weeks of the course were roughly split into a 6-week introduction to compilers, and a 4-week mix of project work and further material. Lectures were a mix of breakout rooms, whiteboarding, and live coding. Asynchronous, online quizzes were given roughly weekly to give students a frequent understanding checks and feedback mechanism.

Figure 1 outlines our intended outcomes for students in the course, when the assignments and lecture/reading topics were assigned, and which outcomes were highlighted each week. PROG was supported every week in the course so we don't write it explicitly. PARSE, CODE, REPR, and TYPE were also supported by the vast majority of project work so we don't necessarily repeat them, though we think the practice with those topics was valuable even in weeks where we weren't introducing new concepts related to those outcomes.

The first four assignments were not part of the large course project, and focused on implementing incrementally increasing subsets of ChocoPy in the style of Ghuloum [3] and as recommended by Karlsson [7].[3] Following these assignments, students completed a group project contributing to a central compiler that started roughly where the individual assignments left off, which we detail in section 2. Students had a final individual oral assessment to test their overall understanding.

[1] The course web page is available at https://ucsd-cse231-w21.github.io/

[2] The institution has a practice of "shopping" courses for the first week or two.

[3] Though we note that, in slight contrast to Karlsson's recommendations that the language used for examples "should be similar to ChocoPy but smaller", we believe we were able to stick to strict *subsets* of ChocoPy.

SPLASH-E '21, October 20, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9089-7/21/10.
https://doi.org/10.1145/3484272.3484961

Abbrev	Description
PARSE	Connect an off-the-shelf parsing library and grammar to code generation
PROG	Program in recursive-descent style Typescript and programmatically generate WASM code
OPT	Describe and identify (though not necessarily implement) control-flow optimizations and lowering surface syntax to an intermediate representation
CLOS	Describe and identify strategies for implementing closures
REPR	Design and implement the representation of a new type of value in an existing compiler
CODE	Design and implement code generation for primitive values, heap-allocated records (e.g. classes with no inheritance), and first-order functions
TYPE	Design and implement type checking rules to add a feature to an existing compiler
DYN	Design and implement code generation for dynamic code loading (e.g. for a REPL or JIT)
SPEC	Read and implement a language specified by a grammar and inference rules
ADD	Identify and resolve likely interactions between new features added to a compiler
PROJ	Work on a group project that can be made public and linked from a resume

Week	Reading and Lecture Topics	Assignment	Outcomes
1	Parser generators, Typescript, WASM, Codegen	Basic Compiler	PARSE, CODE
2	REPLs, WASM Memory, Functions, Datatypes	Basic Compiler	DYN, CODE, SPEC, REPR
3	Static Types and Type Checking	Funcs & Types Compiler	TYPE, SPEC
4	Classes and objects, dynamic heap allocation	Funcs & Types Compiler	TYPE, CODE, DYN
5	Inheritance, nested functions, function inlining	Code Review	CLOS, REPR, TYPE, ADD
6	Closures, nested functions	Classes Compiler	CLOS, REPR, ADD
7	SSA, Liveness	**Project Preferences**	OPT, CODE, PROJ
8	Intermediate representations	**Project Design + Start**	OPT, CODE, PROJ, ADD
9	Control-flow optimizations / **Mob Programming**	**Project Milestones**	OPT, PROJ, ADD
10	Data flow graphs / **Mob Programming**	**Final Code Deadline**	OPT, PROJ, ADD

Figure 1. Course Outcomes and Calendar

1.2 Tools and Technologies

Chocopy. We used ChocoPy, a pedagogic, statically-typed subset of Python [9], as our compiler's source language. Python is familiar to our students and there are off-the-shelf tools are available for Python's syntax [5, 6]. It has an independent **specification** and black-box **implementation** that serve as a source of truth for students when working on their own compilers. In retrospect, we are satisfied with our decision and praise ChocoPy's design.

A Web-based Implementation. We chose to make our implementation run within Web browsers to satisfy two goals: we wanted the language to be interactive (with a visual REPL) and allow student demos to be releasable, compelling, and accessible. To this end, we chose WASM [4] as a target language and TypeScript [8] as an implementation language. WASM is a burgeoning technology, with large existing efforts to develop backends that target it [1, 10].

2 Course Project

For the last four weeks, students worked on course projects. The project was intended to give students the experience of implementing a meaningful feature of a large compiler. The goal was to have the whole class collaborating on different features of one publicly available compiler, which started with the instructors' reference compiler. Figure 2 gives an overview of the project topics, and some of their outcomes and interactions that we detail further in section 2.1.

We suggested the projects and matched students to them based on their preferences. The students formed 17 teams (each 2-3 students), with 15 opting to contribute to the public compiler.[4] The project took 4 weeks and consisted of 4 steps, first choosing preferences and teams, followed by two intermediate week-long milestones, then a final submission (figure 1). At the end of each milestone, students would write and submit a progress report as a Github pull request. The progress report included both passing and pending tests, code updates for their implementation, overall progress since the last milestone, and steps to be accomplished at the next milestone. Then, in the final week of the course each team submitted a final version of their feature.

2.1 Managing Conflicts

A large, collaborative project involves managing conflicts between groups submitting to the same central repository. We had two main strategies for managing conflicts: **mob**

[4]The two groups with Merged as N/A didn't want to manage cross-project collaboration or wanted to work independently.

Feature	Merged	Key Conflicts
Memory Management [MM]	All	**All but Errors/NArgs/Testing**
Lists	All	MM/Strings/Comp/Destruct/Iter
Bignums	All	MM/Lists/Strings/Dicts
For loops/iterators [Iter]	Partial	Strings/Lists/Comp/Destruct
Destructuring Assignment [Destruct]	All	Lists/Strings/Iter/Comp
Strings	All	MM/Dicts/Iter/Bignums
List Comprehensions [Comp]	Partial	Lists/Iter/Destruct
Built-in libraries (NumPy)	None	MM/Lists/Bignums
Dictionaries [Dicts]	All	MM/Strings/Bignums
Closures	All	MM/NArgs
Named Arguments [NArgs]	All	Closures/Types
Type Inference [Types]	None	Errors/NArgs
Web UI/UX	All	Errors/MM
Error reporting [Errors]	All	UX/Types
Testing	All	None
Strings	N/A	N/A
Inheritance	N/A	N/A

Figure 2. Course project topics, whether they merged by the end of the quarter, and key cross-project conflicts

programming, where the instructors led the class in several merges, discussing design tradeoffs and taking suggestions from students, and **asynchronous iteration**, where one project would merge and other groups would update their pull request to match. Occasionally the instructors did additional cleanup or harmonization, and students did some ad hoc organization on Slack, but these two strategies accounted for most merges. Here we detail some anecdotes of notable conflict resolutions.

Memory Management. WASM provides an unmanaged linear memory for use as a heap. The memory management team took on the challenge of building an allocator and collection scheme with an API for other groups to use. This meant that at the milestones, memory management had to be merged either first or last. First, and each other group could make small changes to adapt to the new API, or last, so the memory management team could update all the other groups' code generation before its merge. This was mostly managed through **asynchronous iteration**, where the memory management group took on much of the burden. We wished partway through that this group's API could have been represented as specialized allocation expressions in an intermediate representation (IR), but moving all groups to use an IR was infeasible mid-project.

Strings and Memory Management. The group working on strings changed object initialization to make space for string literals as initial field values. This could complicate how the heap head pointer for allocation was tracked. Together, we derived the idea during **mob programming** of using interned strings to avoid this issue.

Indexing Expressions. Several groups added the Python expression `expr[key]` in separate branches, and we had to pick a single general representation that would work for all of them (including the case where `key` was a slice,[5] which wasn't considered by all groups). Here we decided on the approach as a **mob**, and then groups managed the **asynchronous iteration** after the meeting.

Lesson (Mob Programming). The idea for these sessions came up organically, so we didn't decide on the order of reviewing and merging pull requests beforehand. In some cases, we called on students in the relevant group to briefly assume the role of instructor to explain their approach. We were excited with the level of engagement students had in these sessions. Next time, we would identify and notify groups ahead of time so they could prepare to attend, present, and discuss their code. In addition, with enough such sessions, we could assign each group this role at some point, creating another opportunity for formative assessment for each group.

2.2 Project Outcomes

Ultimately, the resulting compiler (figure 3) represented excellent effort and programming work by students. The final compiler had some issues. Some of these issues were design flaws that weren't caught early enough. Others were incompatible feature interactions that were too time consuming to reconcile. Others were simply a lack of thoroughness and robustness in testing and implementation of a feature.

Some fundamental features suffered from poor ordering of feature implementation by the instructors. For example,

[5] https://docs.python.org/3/tutorial/introduction.html#lists

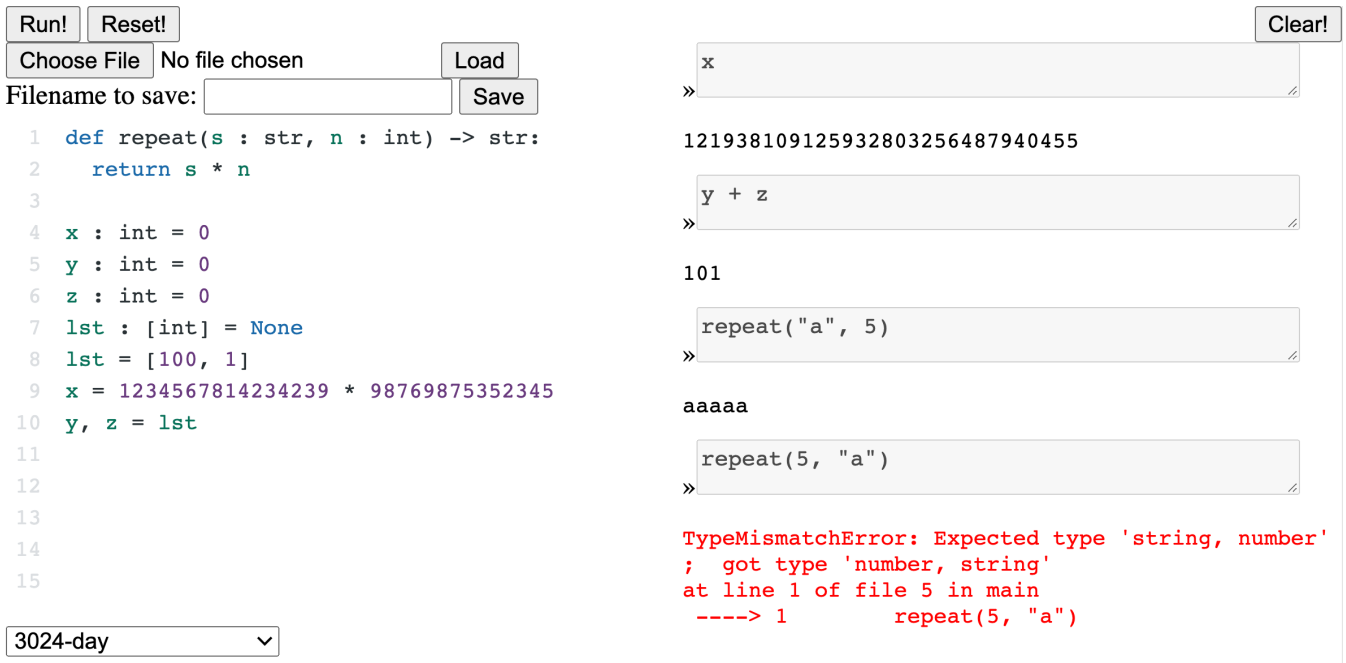

Figure 3. A screenshot of the compiler highlighting destructuring assignment, bignums, function calls, and nicely-rendered static error messages. All of the demonstration expressions on the right are part of the functioning REPL. The source is available at https://github.com/ucsd-cse231-w21/chocopy-wasm-compiler.

`for` loops in Python are designed to work over many different sequence structures through a shared iterator interface. However, when the loops group started their work neither iterators nor lists nor strings were implemented, so the group hacked a custom iterator value to work as `range`.[6] Meanwhile, the lists and strings groups were dealing with issues of memory management and representation, so a good, centralized iterator interface didn't materialize. As a result, `for` loops only work on numeric ranges in the final compiler. List comprehensions had a similar category of issues. Destructuring assignment nearly suffered the same fate (there were no values to destructure when they started), but that group managed to hardcode enough cases for the different data structures to have eventual success.

The closures group implemented new passes over the AST to check for free variables and implement closure creation. However, new AST forms added by other groups were not introduced into all contexts in testing. As a result, new expressions weren't handled in this new AST pass, causing surprising failures for basic features when they appeared within function bodies rather than at the top level, which was the default way to write a test. The relevant groups didn't have time to find and fix all of these issues.

The groups working on types and type inference, and on connecting our compiler to native libraries like NumPy, made useful progress. However, these projects required fundamental changes to the compiler's data structures that were too time consuming to reconcile with each other and the base compiler.

Lessons. We take several lessons from this. First, some features, like the iterator interface, are so central that the instructors may need to declare them by fiat (as they declared that the compiler would have a separate type checker, and that booleans would have a particular representation, and so on). Second, this experience was an important reminder of teaching and emphasizing integration testing. While tests were written for each feature, there was not enough focus on keeping up with testing *feature interactions*. As a result, no one was responsible for making sure new expressions worked in all contexts or with one another.

Of course, the fact that the compiler overall was brittle doesn't mean nothing worked at all. The resulting compiler is a memory-managed language with classes, built-in data structures, and higher order functions that runs interactively in a web browser! And of course, the weaknesses don't necessarily detract from the *learning* experience of contributing to it and seeing various successes and failures of integration. Indeed, the fact that a feature can fail to work is a part of the learning experience as well. So as a vehicle for learning outcomes about the structure of a compiler and how to contribute to one, we found the project effective enough to want to use it again.

[6] With the blessing of the instructors, who perhaps realized their mistake too late.

3 Assessment

Group projects naturally introduce a tension in assessment of individual students versus the group. Since our course (a) required giving grades and (b) fulfills a requirement for students' degree, we were obligated to have a concrete individual assessment for the group projects.

To accomplish this, after the final submissions of group projects, we gave each individual student a follow-up design extension directly related to their project. Their individual assignment was to complete a design document describing how they would approach implementing the extension, and hold a 1-on-1 interview (a kind of oral exam) with us so we could ask clarifying questions.

Sample extensions included: for a student in the group implementing arbitrary-precision integers, how they would add floats to the language; for a student in the group working on lists, how they would implement assignment to slices on lists; for a student working on strings, how they would implement the `format` function, and so on. Each of these was either explicitly, or related to, something they mentioned as un-implemented in their final project submission. The exact prompts we gave to students are in Appendix A.

In order to reduce our workload (48 interviews is a substantial time investment), we reviewed the documents ahead of time for cases where (a) we could easily tell that the style and writing in the reports of different group members differed significantly and (b) there were no problematic open questions that we wanted clarification on. As a result, we notified over half of the students that their oral exams were optional and they'd earned a full score already.

The reports and interviews were a valuable assessment and did find some misconceptions. For example, some students at the end of the class still made first-order vs. first-class mistakes, describing functionality that would only work on e.g. indices given as literal integers in the source rather than expressions that could evaluate to integers. Another category of mistake was not knowing how to decompose a feature across its static and dynamic behavior. These led to productive conversations, and in several cases, we gave students a chance to follow up with a fix after we pointed out an issue to complete their assessment.

4 Feedback and Lessons Learned

4.1 Anecdotal Feedback

We had several pieces of anecdotal feedback[7] from students. Several students remarked in a post course survey that they had a really strong bonding experience in their groups (and noted that this was a special experience during remote instruction due to the pandemic). Another student remarked that WASM came up in a job interview they had during the course and their experience helped them discuss it. By far the large amount of work was the most common *negative* feedback we received, which we struggled to calibrate due to varying backgrounds among the class and our enthusiasm for tackling an ambitious project. These obervations are similar to those made by Coppit in post-course surveys in a software engineering course with 30-person team projects [2].

4.2 The Next Iteration

We present our lessons learned by describing the course as we plan to teach it in the next iteration.

First, in a 10-week schedule, we want to devote *at least* 6 full weeks to the project.[8] While some amount of individual learning is necessary, we are eager to see what groups can accomplish with more time. Based on our experience with mob programming and identifying key concepts through merging student work, we believe that many learning outcomes can be served through the project model without relying on individual assignments. The early part of the course would involve more live programming and explanation of the infrastructure for the project, quickly developing into student groups making contributions, rather than expecting them to each write their own version of the starter compiler.

With more project time comes the ability to introduce more structured development. As described in section 2.1, we struggled when multiple groups had made different decisions in altering interfaces, but also when they felt constrained to existing ones. To alleviate this, we would structure project work with alternating *implementation* and *design* weeks. In implementation weeks, certain cross-component types and interfaces would be fixed, with students only able to implement within provided interfaces, while generating wish-lists for new or changed interfaces. Then design weeks would be focused on reconciling interface changes and pushing through the rote refactorings required to propagate them across projects. Further, design weeks would provide an opportunity for students to review code from other groups and give feedback.

Our class had 48 students, most of whom participated in the large project. This was a relatively small class size compared to typical offerings of the course, which can reach as high as 150. We are unsure if trying multiple large projects each at the 30-50 student scale will be best, since integrating the work of 150 people is a qualitatively different challenge than doing it for 48.

We were satisfied with, and won't substantially change, our assessment strategy, the distribution of the project topics, the way students engaged in mob programming and project development, our choice of technologies, and the use of ChocoPy.

[7] While we have direct quotes and a post-course survey, we report on students' experience in generalities, as we did not collect data about their experience with a formal resarch study or consent to reshare in mind.

[8] We liked the first 5-6 weeks of individual project work our course; at institutions with longer semesters we might leave the first part intact, but in 10 weeks we need to shorten it.

References

[1] Emscripten Contributors. 2015. Building to WebAssembly. Accessed July 15, 2021. https://emscripten.org/docs/compiling/WebAssembly.html

[2] David Coppit. 2006. Implementing large projects in software engineering courses. *Computer Science Education* 16, 1 (2006), 53–73.

[3] Abdulaziz Ghuloum. 2006. An Incremental Approach to Compiler Construction. In *Scheme and Functional Programming*.

[4] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *PLDI*.

[5] Marijn Haverbeke. 2021. lezer-python. https://github.com/lezer-parser/python/. Accessed 13 July 2021.

[6] Marijn Haverbeke. 2021. The Lezer Parser System. https://lezer.codemirror.net/. Accessed 13 July 2021.

[7] Tobias Karlsson. 2021. ChocoPy compiler. Course paper, accessed July 15, 2021. https://fileadmin.cs.lth.se/cs/Education/edan70/CompilerProjects/2020/Reports/Karlsson.pdf

[8] Microsoft. 2021. TypeScript. Accessed July 15, 2021. https://www.typescriptlang.org/

[9] Rohan Padhye, Koushik Sen, and Paul N. Hilfinger. 2019. ChocoPy: A Programming Language for Compilers Courses. In *SPLASH-E*.

[10] The Rust and WebAssembly Working Group. 2021. Rust and WebAssembly. Accessed July 15, 2021. https://rustwasm.github.io/docs/book/

[11] Paul E Young and Donald M Needham. 2013. Using a class-wide, semester-long project to teach software engineering principles. *GSTF Journal on Computing (JoC)* 3, 3 (2013), 1–18.

A Assessment Topics

Here we list the brief assignment descriptions we gave to students, categorized by their group, for their final assessment.

Assignment	**Task**
Bignums	Describe how you could include floats in the representation of numbers
Bignums	Describe how you would make bignums work as arguments to range
Bignums	Describe how dictionary lookup would work with bignum keys
Built-in libraries	Describe how you would add the plus operation on constant numbers and ndarrays
Built-in Libraries	Describe how you would make cross-file constructors work
Closures	Describe how you would avoid reference-wrapping variables unnecessarily
Closures	Describe how you would implement printing a closure's scope in the REPL – e.g. printing the closed-over variables
Closures	Describe how you would implement functions nested within methods in a class body
List Comprehensions	Describe how you would implement dictionary comprehensions
List Comprehensions	Describe how you would implement destructuring binding in comprehensions
List Comprehensions	Describe how you would implement parentheses-comprehensions that create generators
Destructuring Assignment	Describe how you would implement the tuple spread operator on destructuring tuple assignment
Destructuring Assignment	Describe how you would implement chained assignment with destructuring
Destructuring Assignment	Describe how you would implement slicing assignments to lists
Error reporting	Describe how to report index errors on string lookup
Error reporting	Describe how to report errors when calling the None value in function position
Error reporting	Describe how to report recursion errors without an explicit bound in pushStack
Named arguments	Describe how you would handle passing a function with default args as a parameter
Named arguments	Describe how you would handle default arguments that are general expressions
Named arguments	Describe how you would handle keyword arguments

Assignment	Task
For loops/iterators	Describe how you would handle general expressions as the iterable in a for loop
For loops/iterators	Describe how you would handle iterating over strings
For loops/iterators	Describe how you would handle iterating over dictionaries
Web UI/UX	Describe how you would implement pretty printing of dictionaries
Web UI/UX	Describe how you would implement pretty printing of objects
Web UI/UX	Describe how you would implement printing the type of a value along with its contents at the REPL
Type Inference	Describe how you might infer the type of variables initialized to list expressions
Type Inference	Describe how you might infer the type of variables initialized to dictionary expressions
Type Inference	Describe how you might infer the type of bracket-lookup expressions
Inheritance	Describe how you would find the correct field to update with multiple inheritance
Inheritance	Describe how you would find the right method to call with multiple inheritance
Inheritance	Describe how you would implement super method calls with multiple inheritance
Lists	Describe how you would implement the list constructor function
Lists	Describe how you would implement slicing assignment to lists
Memory Management	Describe the API you wish WASM exposed for your garbage collector and how you would use it
Memory Management	Describe one strategy you could use to reduce fragmentation or effectively reuse fragmented space
Memory Management	Describe one way to change codeGen or the collector to safely make some objects' lifetimes shorter
Dictionaries	Describe how you would implement the dictionary constructor with an iterable as an argument
Dictionaries	Describe how you would implement the items() method
Dictionaries	Describe how lookup works with string keys
Dictionaries	Describe how dictionary lookup would work with bignum keys
Strings	Describe how you would convert lists to strings with the str function
Strings	Describe how you would pack characters so that there isn't a separate 4-byte word per character
Strings	Describe how you would convert strings to lists with the list function
Strings	Describe how you would implement format strings.
Testing	Describe how you would extend your fuzzer to generate the literal zero with different frequency as a denominator
Testing	Describe how you would extend your fuzzer to generate terminating loops