[redacted]
[redacted]
Final Paper
[redacted]

minic: A (Somewhat Functional) Mini Compiler in Swift

1 Compiler Overview

My compiler is organized into a Swift package containing two libraries and an executable. The primary library, `MiniCompilerCore`, implements all the major stages of the compiler, as described below, and offers a `compile` function that compiles a given `.mini` source file and several options. The second library is `Minilib`, which compiles to `libmini`, a wrapper for `printf` and `scanf` in C to support the Mini language's `read` and `print` statements. Finally, the `minic` executable is simply a front-end for the core compiler, handling command line argument passing and allowing the compilation of several files with a single command.

1.1 Parsing

Parsing is done using lexers and parsers generated by Antlr4, which supports Swift, and the `mini.g4` grammar. These lexers and parsers are invoked as the first part of the compilation process to generate a parse tree, which is then transformed into an Abstract Syntax Tree by several visitors.

The AST is comprised of several different nodes: `Declaration, Function, Program, Type, Visitors, Expression, Lvalue, Statement, TypeDeclaration`. These are all Swift objects that, in addition to original semantic information, track basic metadata such as the line in the original source that component was generated from. Many of these (such as `Expression`) leverage Swift's fairly powerful enums to represent different cases (Section 3.1.1).

1.2 Static Semantics

Immediately after the AST is created, a Symbol Table is generated for use throughout the compiler. The Symbol Table takes the form of a two-tier lookup table, where each function has a local symbol table that links back to a global symbol table. The global table tracks global variables, functions, and struct declarations (and their fields) while the function tables simply manage their locals.

1.2.1 Type Checking

Type checking is implemented using a recursive descent pattern, with typecheck functions overloaded for each element of the AST. Literals and Identifier expression (with the help of the symbol table) evaluate to concrete types, which are then propagated up the evaluation stack. Errors are printed to standard error and evaluate to the "error" type (to prevent cascading reports), but do not stop the type checking process.

1.2.2 Return Checking

Return checking is very similar to type checking, and also leverages recursive descent. However, only a subset of Mini statements are considered: `block`, `conditional, return`. Return checking guarantees that every function returns by checking for a return or return-equivalent statement along every path. A conditional is return-equivalent if both its blocks contain return-equivalent statements. A block is return-equivalent if any statement in the block is return equivalent. As with type-checking, an error in one function does not stop the checks from running against the remaining functions.

1.3 Intermediate Representation

My compiler uses a fairly typical intermediate representation of a Control Flow Graph in which each node is a Basic Block consisting of a run of non-branching LLVM instructions.

1.3.1 Control Flow Graphs

CFGs are a useful unit of abstraction when compiling a program: each function is somewhat naturally constrained to single entry and exit points. This "bottleneck" creates useful regions to analyze various properties of the code being compiled. When compared to an AST, a CFG is more easily traversed in a manner that matches how the compiled code will run; each block is aware of its possible predecessors and successors. This enables several kinds of data-flow analysis to ascertain, for example, the states of variables throughout the function or whether certain blocks are inaccessible.

In my compiler, each CFG is an object in which all the information pertaining to compiling one function lives. Each CFG also handles the creation of each basic block by "integrating" statements recursively. The result is an array of basic block objects. Similar to the CFG objects, BBs contain an array of LLVM instructions as well as the information relevant to that block (mostly use/def metadata for SSA construction).

1.3.2 LLVM

LLVM is an instruction set for a virtual machine, which brings several benefits. First and foremost, many real constraints (such as register count or calling conventions) can be ignored, making construction of the IR dramatically simpler. Because many important optimizations can be performed directly on LLVM, this allows compiler designers to focus on "best case" optimizations and analysis unconstrained by the minutiae of real platforms. Secondly, and more practically in modern design, new compilers can simply target LLVM (rather than native machine code) while new vendors for architectures can focus on implementing the LLVM to native side, creating a separation of concern for developers and allowing generalized optimization efforts to apply to either party.

As my compiler (optimistically) targets compiling to ARM assembly, the second benefit is largely irrelevant. However, as I discuss in the sections on ARM gen (1.5) LLVM is far easier to work with as an IR than any more constrained instruction set, or any novel one I would have created (and likely bungled). LLVM in my compiler is represented with a Swift enumeration (of available instructions) and some wrapper classes (for ease of use).

1.3.3 SSA

Finally, another major part of the IR is the construction of LLVM in Single Static Assignment form. SSA form requires that all values are defined only once, and never modified; instead, "changes" to a value result in an entirely new value. To handle loops or forks in the definition of a non-SSA value, a special operation ϕ, acts a "join" for several values based on which branch led to the ϕ operation, producing a new unified value.

In my compiler, as in many others, SSA makes several optimizations significantly easier. Unused variables and dead code can be trivially removed by following and iteratively removing SSA values with no uses. Constants are partially propagated as a side effect of SSA construction, and further propagation is easy. It also allows (in the best case) register allocation and rewriting when going to ARM to be dramatically easier.

1.4 Optimizations

Beyond getting code to simply run on a given target, good compilers create faster and more efficient code to the best of their ability. My compiler implements a few minor optimizations.

Useless Code Removal My compiler implements basic useless code removal by iteratively removing non-side-effecting instructions whose results go unused. This eliminates pointless writes and computations.

Constant Propagation While SSA construction propagates trivial constants, constant expressions derived from constant operands can be abstractly evaluated ahead of time. This includes all arithmetic and boolean expressions, and any expressions whose operands become constant through this process.

While this abstract interpretation could be used to eliminate branches whose conditions become constant, limitations in my data structures prevented me from implementing this optimization.

Peephole and Other Optimizations My compiler also manages a few minor custom optimizations. For example, strict translation from LLVM to ARM would create unnecessary register use and `cmp` instructions to handle LLVM's `cond` instruction (which takes a predicate). My compiler coalesces an LLVM `icmp` followed by `branch` into an ARM `cmp` and `b` with the appropriate condition code, skipping the allocation of a register to hold the boolean result of the `cmp` instruction. The compiler also attempts to flatten trivial blocks out of the CFG, removing blocks that contain only a jump and who do not participate in a necessary ϕ operation.

1.5 ARM Code Generation

As with the LLVM IR, I attempted to implement my internal representation of ARM as an enum, creating a kind of domain-specific language within Swift. I map LLVM to ARM by iterating the instructions in each basic block, attempting to minimize register use by rearranging register/immediate pairs to fit ARM's typical register/operand pattern. Where necessary, I insert instructions to force immediates into a register prior to the instruction being translated.

Phi instructions, of course, cannot operate on a real processor by nature of their "branch aware" coalescing operation, so they replaced with `mov` instructions in the prior block to guarantee that by the start of the current block, the phi is effectively already evaluated.

Dealing with the stack on ARM also presents interesting challenges, especially when allocating stack variables and handling functions with more than four parameters. Extra parameters are loaded or pushed (as necessary) with the appropriate instructions, and the corresponding stack cleanup manipulation instructions are inserted at the same time, thanks to a linked-list structure for instructions that allows in-between insertion of instructions.

1.5.1 Register Allocation

Each ARM instruction is augmented with a computed property that returns the source and target registers for that instruction, which is leveraged to build an interference graph for register allocation; the graph is implemented as adjacency-sets (Swift lacks a native graph implementation). Registers are then removed from the graph and "painted" with actual ARM registers. Fixed registers (parameters, return values, etc) are removed first, and then constrained registers, and finally unconstrained registers.

Registers that cannot be painted are "spilled", although issues with re-writing instructions means that this implementation is incomplete. However, only 2 of the current benchmarks spill any registers when fully optimized.

1.6 Other Notes

I have a fair amount of discussion of my own in Section 3, but I also have a few notes to put here. I'm fairly happy with how the code separation for LLVM and ARM turned out; adding new architectures should actually be reasonably easy. Unit testing for the LLVM stages (which runs through XCTest in Xcode or through `swift test`) runs smoothly and in parallel, allowing me to verify my changes did not break the LLVM stage of compilation in less than a minute.

2 Results and Analysis

To profile the results of my compiler, I ran each benchmark 10 times in succession, and averaged the "real" results from `time`. For the LLVM tests, this was performed on my MacBook Pro (Late 2013) while it was left idle, and for the ARM tests this was performed on Cal Poly's Raspberry Pi (# 1). While other users occasionally used the RPi at the same time, none were performing profiling at the same time (they primary were using `gdb`).

Overall my compiler did not perform as anticipated. Because of deep bugs with register allocation, I could only test the full suite of benchmarks and optimization levels with generated LLVM, not ARM. While a visual inspection of the generated LLVM indicated better code, compiling and profiling the resulting binaries on my laptop resulted in performance *degrading* on many benchmarks with optimizations. I am unable to discover why this is; I suspect clang is optimizing the input stack-based LLVM despite being instructed to disable all optimizations (disassembly seems to indicate this).

LLVM instruction count did drop dramatically with SSA construction, and again slightly with the optimizations (Table 1).

The full results are listed in Table 2 in the Appendix. Note that the graphs below use a logarithmic scale; this is so the shorter benchmarks are not lost under the longer ones.

3 Project Reflections

While I am not entirely pleased with my compiler, this was a complex and valuable project, and I wanted to compile (does $\LaTeX$ have a "pun" typeface?) some thoughts on my work.

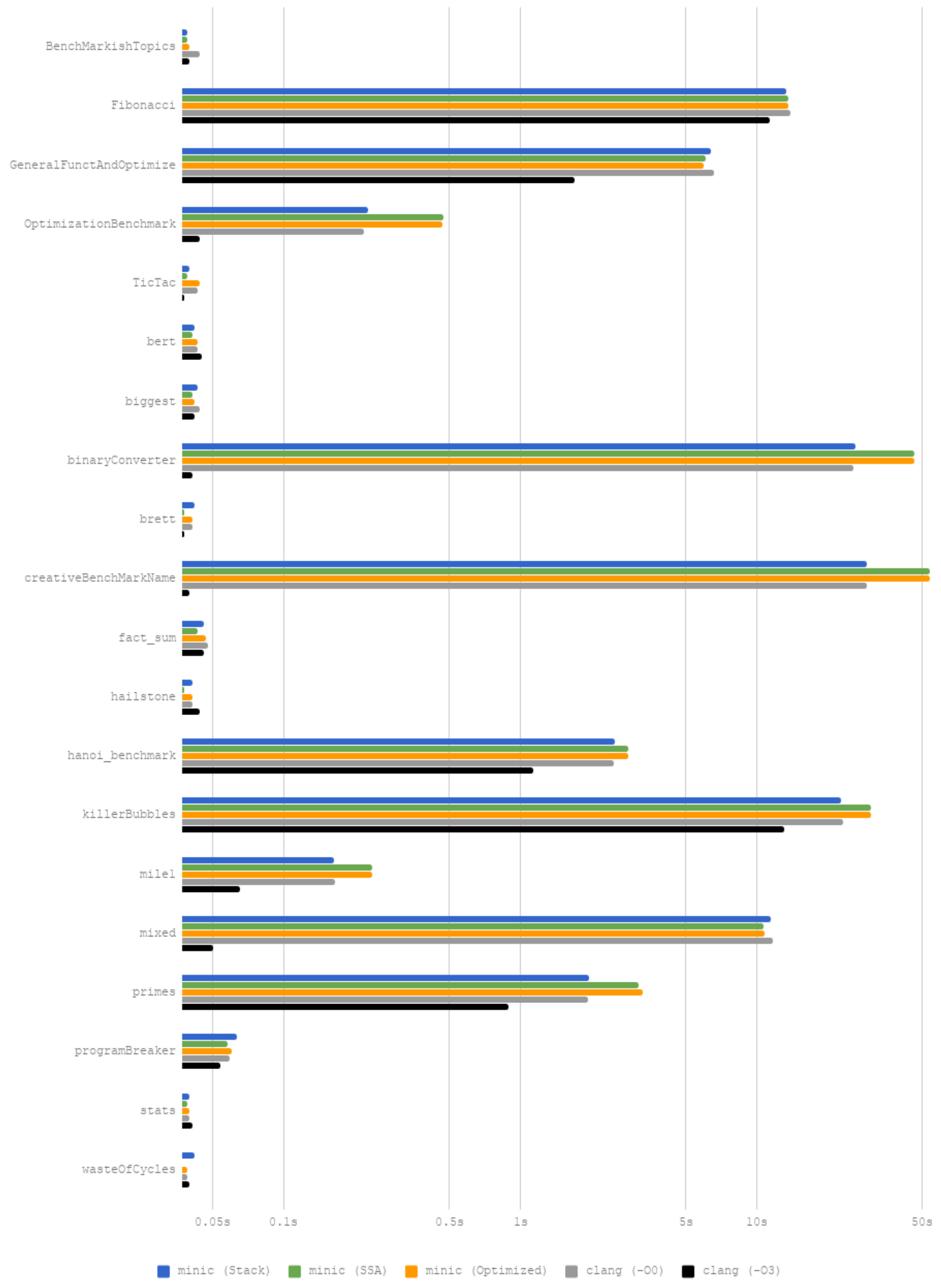

Figure 1: Comparison of `minic` LLVM-derived executables on x86 (2013 MacBook Pro)

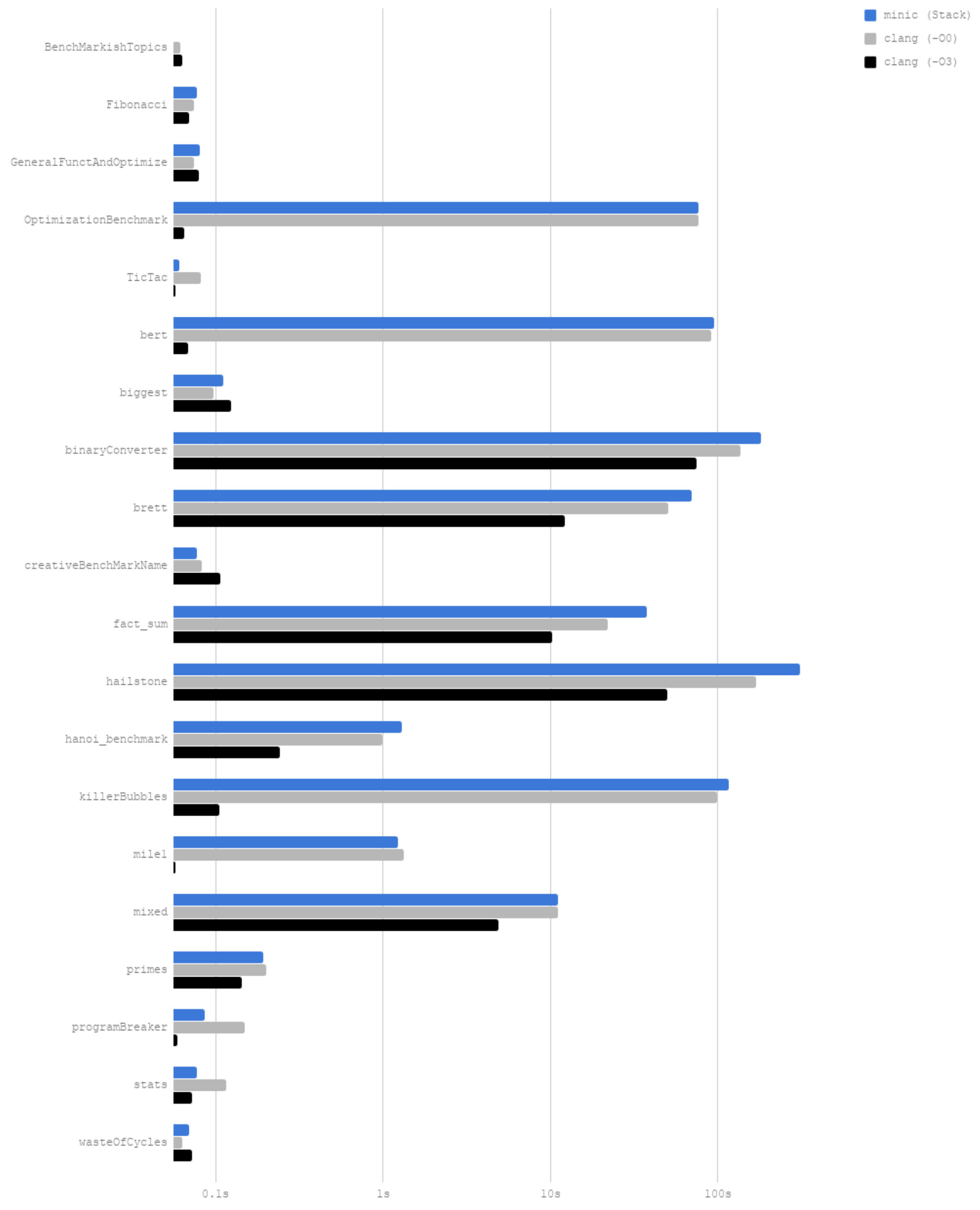

Figure 2: Comparison of `minic` ARM-derived executables on Raspberry Pi

Benchmark	minic (Stack)	minic (SSA)	minic (Optimized)
BenchMarkishTopics	175	127	127
Fibonacci	57	41	41
GeneralFunctAndOptimize	186	134	120
OptimizationBenchmark	1184	555	327
TicTac	544	514	509
bert	859	587	562
biggest	133	87	86
binaryConverter	181	106	106
brett	876	724	692
creativeBenchMarkName	297	181	178
fact_sum	108	70	66
hailstone	81	60	60
hanoi_benchmark	251	186	186
killerBubbles	238	156	153
mile1	106	70	68
mixed	260	173	155
primes	140	90	90
programBreaker	118	77	75
stats	309	200	200
wasteOfCycles	74	49	49

Table 1: Comparison of LLVM instruction counts

3.1 Working with Swift

I originally chose Swift because I wanted a language that supported OO design without the rigidity of Java, and without the complexity of C++. I also wanted to extend my experience with the language, as my limited work with it in the past led me to believe there were several language features that would work well with the project. Overall, it met those requirements, but there were some definite pros and cons to the language.

3.1.1 Positives

Syntax/Type Inference While not the most critical feature of a language, I really grew to enjoy Swift's syntax after a while. Nearly everywhere you wish you could remove symbols or types because they are redundant or obvious, you could. The option of labeling parameters was also convenient in creating self-describing code. Most other syntax features (for in loops, generics) worked exactly as would expect.

Enumerations and Functional Programming Swift's enums allow simple, raw-value enums (as in C/C++), object-like enums (as in Java), and full on functional-style enums (like in SML) using associated values. This worked great for the AST and can be expanded (as I attempted to do) to create strongly typed functional constructs using Swift's equally powerful `switch` statements (which allow pattern matching beyond what SML allowed). Swift also makes Optionals a into a first-class feature of the language, which are quite handy and well integrated with optional chaining and pattern matching. I feel like I could have implemented the CSC 430 project in Swift in a fully functional manner very easily.

Object Extension Swift supports extensions on existing classes (like Objective-C), which allow code to be separated into separate files and existing classes to be easily extended. It also supports computed properties, a common feature of modern languages that is nevertheless handy for avoiding the clutter of getter/setter calls.

Swift Interpreter Swift support a live interpreter mode, which is useful for exploring language features and prototyping. As a compiled language, the interpreter is an unexpected tool, although a bit slow at times.

3.1.2 Negatives

Dealing with References Swift offers, somewhat interestingly, two base types for all values: Object and Struct. Objects are *always* pass by reference, Structs are *always* pass by value. Arrays, and most other built-in collections in Swift, are structs. This means that arrays, sets, and dictionaries are all (by default) pass-by-value copy-on-write, which can be incredibly frustrating to deal with in a compiler (where you may wish several objects had access to the same mutable dictionary).

Structs can be passed by reference to a function using the `inout` qualifier, but you cannot declare properties in a similar manner. This means structures like doubly linked lists *must* use objects (instead of structs) for their nodes to prevent recursive structures.

Mutation Many of the functional parts of Swift encourage or enforce immutable values where possible, and using `let` to declare immutable values is standard style in the language. While this is generally a beneficial system, it makes certain tasks very difficult, such as updating the registers in an instruction that is built using enum values (you have to re-init the value).

Furthermore, Swift will not let you mutate any field of an immutable struct (unless through a `mutating` method), but it *will* let you mutate anything you want about objects declared with `let`, except of course the reference itself. The result is `let` becoming a less consistent version of `const` from C++, a choice I don't quite agree with.

Reference Counting Swift maintains interoperability with Objective-C, and chose to also use reference counting for its garbage collection strategy. While most of the reference counting is automatic, circular links are explicitly the programmer's problem. With any suitably complex data structure, especially one like a compiler's IR with graphs and register-instruction relationships, memory leaks all but guaranteed (or perhaps worse, early deallocation).

The biggest weakness (where's that pun typeface?), however, was the inability to declare that a property must be declared as a `weak` reference in a protocol, which means structures that contain protocol-adhering values cannot convey to their users that a strong reference will create a reference loop. I created wrappers to deal with this, but this felt like a clunky fix.

It is also possible to create a trivial singly-linked list that will crash with a stack overflow upon de-allocation, due to recursive deallocation. Solution? Iterate backward, freeing nodes one at a time, instead of simply throwing away the head.

Generics and Abstract Classes Swift supports generics for the most part as would expect, with two key exceptions: there are no abstract classes, and no generic protocols (which are similar to Java's interfaces). Abstract classes, in typical Swift fashion, should really be re-worked into a protocol if possible. But protocols cannot be generic; instead they require classes that adhere to the protocol to declare an *associated type*. This works fine for many cases, but you cannot use protocols with associated types as types of their own (say you want to declare a protocol that requires a list of values that also adhere to that protocol). The result is a fair amount of frustration with complex generics that is not present in other languages.

Unfinished Language Features The most important issue I had with Swift, was that it is simply not complete yet. While the lack of certain data structures in the standard library is an intentional design choice, some features—like threading, file I/O, and platform consistency —are just not there yet.

Some of the language's more powerful features, like `case` matching in if statements, are also not consistent in their availability. I cannot, for example, check if a value is a certain enum case *and* have a boolean check in a single if statement (I have to nest them). `where` clauses also have inconsistent availability and meaning, from qualifying generics (sometimes) to filtering for loops (but not while loops).

3.2 Data Structures

I have a new found appreciation for data structure design. While time management definitely impeded my completion of a fully functioning compiler, I was completely hamstrung by the end because of design choices I had made earlier in the process. Immutable instructions (by way of Swift enums) made rewriting registers difficult. My register class attempted to get around this by allowing a register to become an "alias" for another register, but this really just created a mess of inconsistencies in how registers reported where they were used.

Similar mistakes were made throughout the compiler; access to my symbol table (and who took ownership of that object) and what it contained underwent several revisions. For a while I had added functionality onto the AST objects to help LLVM generation, but that was ultimately refactored once I realized it made little sense.

3.3 Tests

Good tests are critical for large projects. I was fairly proud of my Swift unit-testing based LLVM regression tests, which would compile each benchmark to LLVM, compile that (with clang) to executables, and then run each one in parallel over several threads. Nothing was more relieving than seeing the green checks after a round of coding.

On the flipside, my ARM tests were abysmal shell scripts. I say abysmal, because the main one incorrectly used old binaries when new ones failed to compile (and hid the errors), reporting incorrectly that my SSA ARM code passed nearly all tests when, in fact, it passed four. I discovered this late at night, just before the final demo day.

3.4 Time Management

Ultimately, the number one culprit for why this project didn't get done to my expectations was my own poor time management. The early parts of the project were easy, and I got lulled into believing I could catch up on some of the later parts. ARM gen with register allocation was very difficult, time consuming, and full of bugs and rewrites. When progress starting slowing down, I made the

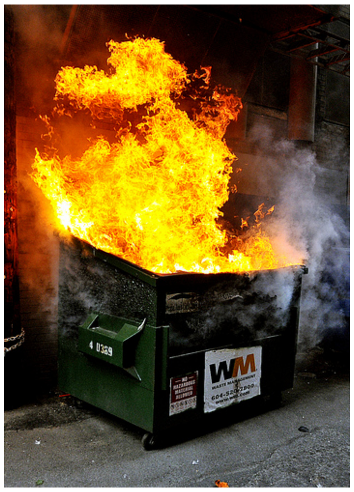

Figure 3: My thoughts on my compiler as of week 10

mistake of pursuing several rounds of "yak shaving" to avoid (or in support of) total rewrites of large chunks of code. Both strategies were fruitless.

4 Closing Thoughts

Compilers are incredible pieces of theoretical computer science and software engineering. While from a black-box perspective my view on what a compiler can do has not changed much, I have a much better appreciation for how they do it. I can confidently say I know Swift very well now. I continue to be impressed at the absolutely insane performance achieved by clang and GCC at maximum optimization levels. Finally, I enjoyed working on this course, even if at the time of this writing, I am not pleased with the finale.

Notes for those who follow:

- Start early, especially at milestone 3 onwards. ARM will take forever to get right.
- Don't get to fancy with your data structures; you won't know what you *really* need until a later milestone, so don't architect a beautiful solution that is totally useless next week.
- Test your tests.

Finally, I wish the include with my submission (separately) my AST in Swift, its visitors, and how to get Antlr to work with Swift (as of this writing) should anyone else wish to pursue a non-Java path in the future. I conclude with Figure 3. Thank you for reading, and enjoy the break.

5 Appendix

Benchmark	LLVM					ARM				
	minic			clang		minic			clang	
	Stack	SSA	Opt.	-O0	-O3	Stack	SSA	Opt.	-O0	-O3
BenchMarkish...	.0039	.0039	.004	.0044	.004	.0056	TF	TF	.0062	.0063
Fibonacci	1.328	1.356	1.355	1.388	1.134	.0077	11.865	11.924	.0074	.007
GeneralFunct...	.6382	.6083	.5924	.66	.17	.0081	NC	NC	.0074	.008
Optimization...	.0226	.0473	.0466	.0219	.0044	7.668	NC	NC	7.643	.0065
TicTac	.004	.0039	.0044	.0043	.0038	.0061	NC	NC	.0082	.0058
bert	.0042	.0041	.0043	.0043	.0045	9.485	NC	NC	9.180	.0069
biggest	.0043	.0041	.0042	.0044	.0042	.0112	.0083	TF	.0097	.0124
binaryConverter	2.607	4.616	4.615	2.562	.0041	17.990	NC	NC	13.734	7.438
brett	.0042	.0038	.0041	.0041	.0038	6.958	NC	NC	5.047	1.223
creative...	2.907	5.383	5.394	2.912	.004	.0078	NC	NC	.0083	.0107
fact_sum	.0046	.0043	.0047	.0048	.0046	3.789	.0096	TF	2.201	1.025
hailstone	.0041	.0038	.0041	.0041	.0044	31.160	.0059	.0092	17.034	4.978
hanoi_benchmark	.2502	.2852	.2859	.2468	.1134	.1297	NC	NC	.0997	.0241
killerBubbles	2.267	3.034	3.032	2.303	1.302	11.684	NC	NC	9.887	.0105
mile1	.0163	.0235	.0235	.0164	.0065	.1222	NC	NC	.1327	.0058
mixed	1.137	1.061	1.072	1.164	.005	1.109	NC	NC	1.109	.4884
primes	.194	.3143	.3285	.1935	.0887	.0194	NC	NC	.02	.0143
programBreaker	.0063	.0058	.006	.0059	.0054	.0086	.0159	TF	.015	.0059
stats	.004	.0039	.004	.004	.0041	.0077	NC	NC	.0116	.0072
wasteOfCycles	.0042	.0037	.0039	.0039	.004	.007	NC	NC	.0063	.0072

Table 2: Benchmark Runtime Matrix. NC: did not compile. TF: compiled, but failed tests.