

# 15-411 Compiler Design, Fall 2020

## Lab 5

Compilers Due: 11:59pm, Thursday, December 3, 2020  
Report due 11:59pm, Tuesday, December 8, 2020

### 1 Introduction

The goal of this lab is to implement optimizations for the language L4, which remains unchanged from Lab 4. Your goal is to minimize both the running time and code size of the executable generated by your compiler on a set of benchmarks. This includes an *unsafe* mode in which your compiler may assume that no exception will be raised during the execution of the program (except due to `assert`). This affects operations such as integer division, arithmetic shift, array access, and pointer dereference.

### 2 Preview of Deliverables

In this lab, you are not required to hand in any test programs, since there is no change in language specification. Instead, we will be testing your compiler's optimizations on a suite of benchmark tests created by the course staff. We offer a choice of many optimizations you can implement, and the benchmarks are designed to resemble realistic programs. As in real life, some may respond better to particular optimizations than others. In addition to the benchmarks, we will be testing the correctness of your compiler by running it on the test suites from Labs 1-4, so you must be careful to maintain the semantics of the L4 language when optimizing.

You are required to turn in a complete working compiler that translates L4 source program into correct target programs in x86-64. In addition, you have to submit a PDF file which describes and evaluates the optimizations that you implemented.

### 3 Compilation to Unsafe Code

The `--unsafe` flag to your compiler allows it to ignore any exceptions that might be raised during the execution of the program except ones due to `assert`. This means you can eliminate some checks from the code that you generate. You are *not* required to eliminate all (or, indeed, any) checks, but it will make your compiled code slower if you do not take advantage of this opportunity at least to some extent.

Note that unsafe means that you do not have to perform any runtime checks, not that you can generate an incorrect executable. :) By eliminating the runtime checks you will be more able to compare the efficiency of your generated code against gcc and others in the class.

In addition to the `--unsafe` flag, your compiler must take a new option, `-On`, where `-O0` means no optimizations, and `-O1` performs the most aggressive optimizations. One way to think about this is that `-O0` should minimize the compiler's running time, and `-O1` should prioritize the emitted code's running time and code size. We suggest using some thresholds to decide between the two as the default behavior, if no flag is passed. We will pass the `-O1` flag when timing your compiler on the benchmarks. The `-O0` flag is mainly for your own debugging purposes.

## 4 Optimizations

In the following sections, we will provide you with list of suggested analysis and optimization passes you can add to your compiler. This is a long list and we obviously do NOT expect you to complete all of the optimizations. We suggest that you pick the optimizations that you are most interested in, and do enough optimizations so that your compiler is competitive with the `cc0` reference compiler and `gcc -O1`. We list the course staff recommended difficulty and usefulness rating (your experience may vary) of optimizations to help you decide which passes to implement first.

If you have already implemented any of the optimizations, you may revisit and describe them, empirically evaluate their impact, and improve them further. In this case, your report should contain a description of any improvements you made.

Feel free to add other optimizations and analyses outside of this list as you see fit, although we strongly recommend first completing basic ones before you go for more advanced ones. That said, it is a good idea to consult the course staff first to ensure that your planned optimizations are feasible to complete within three weeks.

There is abundant literature on all the following optimizations, and we have listed some good resources that might be helpful for this lab. We specifically recommend the Dragon Book (Compilers: Principles, Techniques, and Tools, 2nd Edition), the Cooper Book (Engineering a Compiler, 2nd Edition), and the SSA Book (SSA-Based Compiler Design), all of which have great sections on compiler optimizations. Additionally, the recitations, lecture slides, and lecture notes are great resources. We also encourage you to read relevant papers and adapt their algorithm for your compiler, as long as you cite the source.

### 4.1 Analysis Passes

Analysis passes provide the infrastructure upon which you can do optimizations. For example purity/loop/alias analysis computes information that optimization passes can use. The quality of your analysis passes can affect the effectiveness of your optimizations.

#### 1. Control flow graph (CFG)

Difficulty: ★☆☆☆☆      Usefulness: ★★★★★

Almost all global optimizations (intraprocedural optimizations) will use the CFG and basic blocks. We recommend implementing CFG as a standalone module/class with helper functions such as reverse postorder traversal and splitting critical edges.

#### 2. Dataflow Framework

Difficulty: ★★☆☆☆      Usefulness: ★★★★★☆

The Dataflow Framework we made you do in the l2 checkpoint is not only useful for liveness analysis, but also for passes such as alias analysis, partial redundancy elimination (which

uses 4 separate Dataflow passes, see section below), among others. You probably want your Dataflow framework to work with general facts (a fact could be a temp/expression/instruction, etc.).

### 3. Dominator Tree

Difficulty: ★★☆☆☆      Usefulness: ★★★★★

*Resources: SSA Recitation Notes*

You can build a Dominator Tree on top of your CFG. The Dominator Tree will be useful for constructing SSA, loop analysis, and many other optimizations.

### 4. Single Static Assignment (SSA)

Difficulty: ★★★★★☆      Usefulness: ★★★★★

*Resources: SSA Recitation Notes*

A program in SSA form has the nice guarantee that each variable/temp is only defined once. This means we no longer need to worry about a temp being redefined, which makes a lot of optimizations straightforward to implement on SSA form, such as SCCP, ADCE, Redundant Safety Check Elimination, among others. In fact, modern compilers such as LLVM uses SSA form for all scalar values and optimizations before register allocation. Your SSA representation will need to track which predecessor block is associated with each phi argument.

### 5. Purity Analysis

Difficulty: ★☆☆☆☆      Usefulness: ★★☆☆☆

Purity analysis identifies functions that are *pure* (*pure* can mean side-effect free, store-free, etc.), and can enhance the quality of numerous optimization passes. This is one of the simplest interprocedural analysis you can perform.

### 6. Loop Analysis

Difficulty: ★★☆☆☆      Usefulness: ★★★★★☆

*Resources: LLVM Loop Terminology*

A Loop Analysis Framework is the foundation of loop optimizations, and is also useful for other heuristics-based optimizations such as inlining and register allocation. Generally, you will do loop analysis based on the CFG, and identify for each loop its header block, exit blocks, nested depth, and other loop features. You might also consider adding preheader blocks during this pass.

### 7. Value Range Analysis

Difficulty: ★★★★★☆      Usefulness: ★★☆☆☆

*Resources: Compiler Analysis of the Value Ranges for Variables (Harrison 77)*

Value Range Analysis identifies the range of values a temp can take on at each point of your program. You can use an SSA-based or dataflow-based approach. Value Range Analysis can make other optimizations more effective, such as SCCP, Strength Reduction, and Redundant Safety Check Elimination.

## 4.2 Optimization Passes

Below is a list of suggested optimization passes. All these optimizations are doable - they have been successfully performed by students in past iterations of this course.

### 1. Cleaning Up Lab3 & Lab4

Difficulty: ★★☆☆☆      Usefulness: ★★★★★

Before performing global optimization passes, we suggest inspecting the x86 assembly code output of your compiler as you did for l5 checkpoint, and finding opportunities for improvement. For example, you would want to optimize for calling conventions in lab3 (try not to push/pop every caller/callee register), and you would want to make use of the x86 *disp(base, index, scale)* memory addressing scheme to reduce the number of instructions needed for each memory operation in lab4. Another common mistake is a poor choice of instructions in instruction selection (try comparing your assembly to gcc/clang output), or fixing too many registers in codegen and not making full use of your register allocator.

### 2. Strength Reduction

Difficulty: ★★☆☆☆      Usefulness: ★★★★★☆

*Resources: Division by Invariant Integers using Multiplication (Granlund 91)*  
*Hacker's Delight 2nd Edition Chapter 10*

Strength reduction modifies expressions to equivalent, cheaper ones. This includes unnecessary divisions, modulus, multiplications, other algebraic simplifications, and memory loads. Though simple to implement, this optimization can bring a huge performance improvement (a division/modulus takes dozens of cycles on a modern CPU). Getting the magic number formulas for division and modulus is tricky, and we recommend you read the above resources, or look at how GCC/LLVM implements strength reductions.

### 3. Peephole & Local Optimizations

Difficulty: ★★☆☆☆      Usefulness: ★★★★★☆

Peephole and local optimizations are performed in a small window of several instructions or within a basic block. Similar to strength reductions, these are easy to implement but can bring a large performance improvement. We recommend comparing your assembly code to gcc/clang assembly code to find various peephole opportunities and efficient x86 instructions.

### 4. Improved Register Allocation

Difficulty: *varies*      Usefulness: ★★★★★

*Resources: Pre-spilling - Register Allocation via Coloring of Chordal Graphs (Pereira 05)*  
*Live Range Splitting - Lecture notes on Register Allocation*  
*SSA-based Register Allocation - SSA Book Chapter 17*

A good register allocator is essentially for code optimization, and below we provide a list of possible extensions (ordered roughly in increasing difficulty) to the register allocator we had you build for the L1 checkpoint, which is far from perfect. We highly recommend at least implementing coalescing, and the rest is up to you.

- (a) **Coalescing** We recommend optimistic coalescing, which integrates seamlessly into the graph coloring approach taught in lecture. However, there is abundant literature in this area so feel free to explore other coalescing approaches.

- (b) **Heuristics for MCS and Coalescing** Using some heuristics to break ties in Maximum Cardinality Search and decide the order of coalescing might enhance the quality of your register allocator.
- (c) **Pre-spilling** Pre-spilling identifies maximum cliques in the graph and attempts to pick the best temps to spill before coloring the interference graph. You can also integrate this with your current post-spilling approach.
- (d) **Live Range Splitting** The naive graph coloring approach assigns each temp to the same register or memory location throughout its whole lifetime, but if the temp has “lifetime holes” between its uses, one can split its live range to reduce register pressure, especially at the beginning and ending of loops. This optimization is more naturally integrated with a linear scan register allocator, but is still possible with a graph coloring allocator. See the lecture notes on register allocation for details.
- (e) **Register Allocation on CSSA** Doing register allocation and spilling on SSA might be faster and more effective, and with SSA you get a chordal interference graph. However, it turns out that getting out of SSA is a bit difficult after doing register allocation on SSA. You should probably spend your time doing some of the other (more interesting and fun) optimizations if you haven’t already decided to do this. If you’re still interested, the paper *SSA Elimination after Register Allocation* might be useful. Warning: this could be challenging to get right.

## 5. Code Layout

Difficulty: ★★☆☆☆      Usefulness: ★★☆☆☆

As we saw in the guest lecture, optimizations for code layout include deciding the order of basic blocks in your code, minimizing jump instructions and utilizing fall throughs, and techniques such as loop inversion.

## 6. Sparse Conditional Constant Propagation (SCCP)

Difficulty: ★★★☆☆      Usefulness: ★★★☆☆

*Resources: Constant Propagation with conditional branches (Wegman and Zadeck)*

It is possible to do local constant propagation within basic blocks, but we recommend this SSA-based global constant propagation approach. Additionally, SCCP can also trim dead conditional branches that will never be visited. SCCP might not bring a large improvement in code performance, but would significantly reduce code size and improve readability.

- (a) **Copy Propagation** One related optimization is copy propagation, which is straightforward to implement on SSA, and can also serve to eliminate redundant phi functions. Note that much of copy propagation’s functionality is covered by register coalescing, and aggressively doing copy propagation might increase register pressure. However, copy propagation can serve to reduce the number of instructions which speeds up subsequent passes and makes your code easier to debug.

## 7. Aggressive Deadcode Elimination (ADCE)

Difficulty: ★★☆☆☆      Usefulness: ★★★☆☆

Similar to SCCP, deadcode elimination is made easier by SSA, and can bring improvement to both code size and performance. ADCE can be made more effective by purity analysis.

## 8. Partial Redundancy Elimination (PRE)

Difficulty: ★★★★★☆      Usefulness: ★★★★★★

*Resources: Dragon Book Section 9.5, or Cooper Book Section 10.3*

PRE eliminates partially redundant computations, and provides the additional benefits of Common Subexpression Elimination (CSE) and Loop Invariant Code Motion (LICM). The latter is especially important to reduce loop execution overhead. You can implement the SSAPRE algorithm, but we recommend the simpler alternative using 4 dataflow passes which you read about in the Dragon Book or Cooper Book. Your dataflow framework from L2 checkpoint will come in handy here. An alternative to implementing PRE is to implement CSE and LICM as 2 separate passes.

- (a) **Global value numbering (GVN)** GVN can identify equivalent computations in the code, and can either be a standalone pass or be incorporated into PRE to eliminate more redundant computation. Note that GVN might eliminate some expressions that CSE cannot.

## 9. Function Inlining

Difficulty: ★★☆☆☆☆      Usefulness: ★★★★★☆☆

Inlining a function can reduce the overhead of a call and potentially open up opportunities for more optimizations. However, it can bring problems such as increased code size and register pressure. Choosing which functions calls to inline is often a tradeoff between code size and performance, and you will need some good heuristics. For example, common heuristics include size of the functions, and loop depth of the function call.

## 10. Tail Call Optimization (TCO)

Difficulty: ★★☆☆☆☆      Usefulness: ★★☆☆☆☆

*Resources: LLVM TailRecursionElimination pass*

TCO turns recursive calls at the end of functions into jumps to reduce the overhead of function calls. You can also perform accumulation transformations on tail-call expressions when required. You might also find other benefits of turning recursive calls into jumps.

## 11. Redundant Safety Check Elimination

Difficulty: ★★☆☆☆☆      Usefulness: ★★☆☆☆☆

You can implement an SSA-based or dataflow-based approach to eliminate redundant null-checks and array bounds-checks when dereferencing pointers or accessing arrays at runtime. This optimization is specifically tailored towards speedup in safe mode, as these checks can be removed entirely when running with `--unsafe`.

## 12. Loop Optimizations

Difficulty: *varies, hard in general*      Usefulness: *useful for programs abundant with loops*

- (a) **Loop unrolling/tiling/fusion/interchange/...** You need good heuristics to perform these optimizations, and their effectiveness is target specific (dependent on the hardware). Your loop analysis framework will come in handy here.
- (b) **Induction Variable Elimination (IVE)** You need to first perform Induction Variable Detection which detects induction variables in a loop, and dependence analysis. Then you could perform strength reduction, scalar replacement, and deadcode elimination

based on the induction variables. We recommend doing SSA-based IVE. See the lecture slides for more details.

### 4.3 Advanced Analysis and Optimization Passes

Below we list some optimizations that might be beyond the scope of this project - they are either too hard, or might not affect your score enough to justify the time investment. We only recommend you to implement these once you have most of the optimizations in the above section working.

#### 1. Alias Analysis

Difficulty: ★★★★★ Usefulness: ★★★★★

*Resources: Andersen's or Steensgaard's Points-To Analysis*

*Making context-sensitive points-to analysis practical for the real world (Lattner 07)*

We recommend Andersen's or Steensgaard's approach as it is simpler to implement, though you could also refer to Lattner's paper and LLVM's cheaper alias analysis approach. Note that you are allowed to assume the *strict aliasing rule* of C, that the function pointer arguments in a function are assumed not to alias if they point to fundamentally different types. Alias analysis will enhance the quality of many of your optimizations, including ADCE, PRE, redundant store elimination, instruction scheduling, among others.

#### 2. Interprocedural Optimizations

Difficulty: *varies, hard in general* Usefulness: *depends*

Most optimizations in the above section are intraprocedural, but some passes such as register allocation and alias analysis, can be made more effective when applied across functions. Interprocedural Optimizations generally involve traversing the call graph.

#### 3. Vectorization using Streaming SIMD Extensions (SSE)

Difficulty: ★★★★★ Usefulness: *useful for programs with a lot of parallelism*

You can take advantage of the X86 SSE, SSE2, or AVX-512 extensions to vectorize loops, like *gcc -O3* does. This can also be an interesting project for lab6.

#### 4. Instruction Scheduling (Software Pipelining / Hyperblock or Trace Scheduling)

Difficulty: ★★★★★ Usefulness: *depends on the program and the processor*

Code scheduling involves moving instructions around to increase instruction level parallelism and reduce pipeline stalls. You might also perform if-conversions to form larger blocks. These optimizations are very tricky to get right and are target-specific.

## 5 Testing

As you are implementing optimizations, it is extremely important to carry out **regression testing to make sure your compiler remains correct**. We heavily recommend that your optimizations be modular, and that correctness does *not* depend on a particular previous optimization. We will call your compiler with and without the `--unsafe` flags at various levels of optimization to ascertain its continued correctness, though your performance will be evaluated primarily through our benchmarks.

To enable you to perform more compiler optimizations, we increased `COMPILER_TIMEOUT` of the autograding harness to 6 seconds. We also increased the `RUN_TIMEOUT` of the executable produced by your compiler to 120 seconds.

To help you test your performance, you'll see some new files in the `dist` repository:

- `tests/bench/`, which contain the benchmark programs.
- `timecompiler`, a script which counts the cycles of your compiler on these benchmarks.
- `score_table.py`, a script which you can use to generate Notolab-like score tables. You can also see the times and code sizes of the executables produced by `cc0` and `gcc`, and the formula we use to compute your multiplier on Notolab. Read the comments within `score_table.py` on how to use this script.

To use the `timecompiler` script, you should follow these steps:

1. Ensure your compiler supports `--unsafe` and `-O1`.
2. If you wish, add additional benchmarks to the benchmark folder.
3. Run `../timecompiler` from your compiler's directory. `timecompiler` accepts the same flags that `gradecompiler` does – however, we don't recommend running the benchmarks in parallel.

Here is a cheatsheet of useful commands:

`timecompiler bench`:

this will time your compiler on the benchmarks in the `../tests/bench` folder, and print out the cycles and code sizes for each benchmark

`timecompiler -q --autograde`:

the `-q` flag suppresses unhelpful output, and `--autograde` will print a json array of the cycles and code sizes at the end, which you can pass to `score_table.py`

`bin/c0c -ex86-64 -O1 --unsafe ../tests/bench/daisy.l4`:

this generates a `.s` file from your compiler

`gcc -m64 -no-pie ../runtime/run411.o ../tests/bench/daisy.l4.s`:

using the `.s` file, you can link to our runtime file to generate an executable

`gcc -m64 -no-pie ../runtime/bench.o ../tests/bench/daisy.l4.s`:

alternatively, you could link to `bench.o` to generate an executable that will run the benchmark numerous times, and print out the average of the  $k$  best times

`gcc -O1 -fno-asynchronous-unwind-tables -S ../tests/bench/unsafe/daisy.c`:

this uses `gcc` to generate a `.s` file in the current directory, which you can use to compare your own assembly against

`gcc -O1 ../runtime/run411.o ../tests/bench/unsafe/daisy.c`:

alternatively, you can let `gcc` directly generate an executable

## 6 Deliverables and Deadlines

For this project, you are required to hand in a complete working compiler for L4 that produces correct target programs written in Intel x86-64 assembly language, and a description and assessment of your optimizations. The compiler must accept the flags `--unsafe` and `-On` with  $n = 0, 1$ . When

we grade your work, we will use the `gcc` compiler to assemble and link the code you generate into executables using the provided runtime environment on the lab machines.

Note that this time we will not just call the `_c0_main` function in the assembly file you generate, but also four other internal functions in order to obtain cycle counts that are as precise as possible. These are `_c0_init`, `_c0_prepare`, `_c0_run`, and `_c0_checksum`, each corresponding to their un-prefixed counterparts in the benchmark source. Given this, it is critical that your code follow the standard calling conventions and function naming conventions from Labs 1–4 for these functions.

## Compiler

The sources for your compiler should be handed in via Github as usual, and *must* contain documentation that is up to date. Particularly, your compiler should document each of your performed optimizations in both a `README` file and the source itself. The course staff *will* be reading your code as part of the submission for this lab. You may use up to five late days for the compiler.

Compilers are due **11:59pm on Thursday, Dec 3rd, 2020**.

## Project Report

The project report should be a PDF file of approximately 4–5 pages (possibly more, particularly with figures), and should be handed in on Gradescope. Your report should describe the effect of `--unsafe` as well as your optimizations and other improvements, and assess how well they worked in improving the code, over individual tests and the benchmark suite.

At the *absolute minimum*, your project should present a description and quantitative evaluation of the optimizations you performed at the `-O0`, `-O1`, and default levels. A good report must also discuss the way your individual optimizations *interact*, backed up by quantitative evidence. (Tables are a good idea. Graphs are an even better idea.) Make sure to carefully document how you got your numbers; someone with access to your code should, if they're willing to buy whatever hardware and operating system you were using, be able to replicate your results. A good report should also spend some time describing the effect of individual optimizations on the code you produce.

Your report should contain a specific commit hash for the course staff to review<sup>1</sup>, and a comprehensive descriptions of where in your source files each optimization you have described is implemented. If you use algorithms that have not been covered in class, cite any relevant sources, and briefly describe how they work. If the algorithms have been covered in class, cite the appropriate lecture notes or paper, and focus on any implementation choices you made that are not described in those resources.

Other (optional) discussions that might be included in a high-quality report include:

- Effects of the ordering of different optimization passes.
- Time versus space tradeoffs in emitted code.
- Effects of various optimizations on the running time of your compiler.
- Examples of programs that your optimizations would interact particularly well with.
- Examples of programs that your optimizations would interact particularly poorly with.

---

<sup>1</sup>It is not necessary that this be the same commit hash that you submit to Notolab. In fact, we encourage you to perform any code-cleanup that may be required.

Project reports are due on **11:59pm on Thursday, Dec 8th, 2020.**

**Late days:** The late day rules for written assignments apply. You can only use late days if both team members have late days left. If you use late days, then both team members will lose late days.

## Grading

This assignment is worth 150 points. Lab5 checkpoint is worth 20 points. The written report is worth 30 points. The remaining 100 points will be based on the correctness of your compiler and on the performance of your emitted code relative to our benchmarks, as reflected by your Notolab score.

Your performance will be measured as a factor of how far between -00 and -01 you are, and of course, scaled accordingly. Achieving true parity with -01 in the course of a single semester is a tremendous feat, and we offer the comparison primarily to give you a sense of the bigger picture. You will be scored not only based on how fast the code generated by your compiler is, but also by the code size. Note that the running time is weighed much more heavily than code size in your score.

We will run your compiler numerous times in all of the optimization modes, and take the  $k$ -best times of all of these. We will also use linux `size` command to determine the code size of the executable generated by your compiler, and compare to the code size of the `cc0` reference compiler and `gcc`. We will use the benchmark score as a multiplier for your correctness score, and we derive your multiplier by averaging your score over all of the benchmark tests. The scheme is designed so that you do not have to exceed -00 on all tests, and may benefit from a score greater than 1 if your optimizations provide excellent speedup in certain cases. However, you will incur a penalty of -10% for every benchmark which failed to execute correctly or within the time limit on any of the optimization modes.

The exact formula for each test is as follows:  $t_c$  is the average of the  $k$ -best times from your compiler,  $t_0$  and  $t_1$  denote the times of the `cc0` reference compiler using -00 and -01 respectively. Similarly,  $s_c$  is code size of the executable produced by your compiler, and  $s_0$  and  $s_1$  denote the code sizes of the `cc0` reference compiler using -00 and -01 respectively. The  $u$  variables denote the same times or code sizes, but with `--unsafe` and using `gcc` as reference. **Both  $P_s$  and  $P_u$  are clamped between 0 and 2.5.**  $T$  is our benchmark suite, and  $M$  is your multiplier for the lab. For running time of a benchmark:

$$P_s = 1 - \frac{t_c - t_1}{t_0 - t_1} \quad P_u = 1 - \frac{u_c - u_1}{u_0 - u_1} \quad P_{time} = \frac{P_s + P_u}{2}$$

For code size of a benchmark:

$$P_s = 1 - \frac{s_c - s_1}{s_0 - s_1} \quad P_u = 1 - \frac{u_c - u_1}{u_0 - u_1} \quad P_{size} = \frac{P_s + P_u}{2}$$

Overall score:

$$P_{bench} = 0.8 * P_{time} + 0.2 * P_{size} \quad M = \frac{\sum_{bench \in T} P_{bench}}{|T|}$$

Your final score is then  $MC$ , where  $C$  is your correctness score from running the suites from Labs 1-4 on Notolab. We will be running your compiler both with and without the `--unsafe` flag. We will run `--unsafe` on tests with directives `//test return i`, `//test typecheck`, `//test abort`,

`//test error`, and `//test compile`. We will run `--safe` on tests with directives `//test div-by-zero` and `//test memerror`.

You will be able to get **extra credit** on this assignment! If your final score is above 100 points, you will receive  $\min(20, \frac{MC-100}{2})$  extra credit points for building an excellent optimizing compiler.

## 7 Tips and Hints

1. You have less than three weeks to work on this lab, and implementing optimizations and SSA is a lot of work, so start early!
2. The Godbolt Compiler Explorer ([godbolt.org](http://godbolt.org)) is a really helpful tool for comparing your compiler against gcc/clang.
3. As in the Lab 5 checkpoint, you should make a habit of closely inspecting the assembly outputted by your compiler, comparing it to gcc/clang's output, identifying inefficiencies in your code, and thinking about the possible optimizations to address those inefficiencies.
4. That being said, optimizing for a specific line in a specific function in a specific benchmark might not be a good strategy, as it is unlikely to affect your score by much. Global optimizations that eliminates efficiencies across the board are likely a better investment of your time.
5. Since you will likely perform most optimizations on SSA form or some other IR form, compiler utilities and flags to print out code in that IR can be really helpful for debugging, as is using dot to generate visual representations of control flow graphs (you can download graphviz here: <https://graphviz.org/>)
6. The ordering of optimizations and analysis passes can be extremely important, and you should explore how different optimizations interact. It is often worth it to perform a certain pass multiple times (before and after related passes).
7. If testing locally on a Mac outside of a docker container, you may get a slightly worse code size score than you would on autolab or in a docker container. This is because native MacOS uses the Mach-O executable format which rounds the size of the code segment up to a multiple of 4096 whereas the Linux ELF format does not round up at all. We strongly recommend testing for code size inside of a docker container on MacOS – otherwise, you will not be able to measure code size changes less than 4096 bytes.
8. Don't name any local folders `bench`. Our autograder defaults to searching for benchmarks