CS4120/4121/5120/5121—Spring 2026
Programming Assignment 6
Optimization
Due: Monday, May 11, 4:30PM

The goal of this assignment is to improve the quality of your code by implementing high-quality register allocation and various optimizations in your compiler, as well as extending your compiler to support Rho, an extension of Eta. In addition to code, your compiler will generate CFG diagrams showing the effects of various optimizations. You will also submit test programs that show off the effectiveness of your optimizer. Using various benchmark programs, we will compare the performance of your compiler against the compilers produced by other groups, and the compiler that has the most effective optimizations will earn a bonus.

0 Changes

- Added SSA requirement for groups with CS 5120 students.
- Updated submission requirements.
- Added AI policy.
- Added section on SSA specific additions to the IR.

1 Instructions

1.1 Grading

Solutions will be graded on documentation and design, completeness of the implementation, correctness, and style. 10% of the score is allocated to whether bugs in past assignments have been fixed.

For this assignment, you may use no more than two slip days, so that the course staff has enough time to grade your project.

1.2 Partners

You will work in a group of 3–4 students for this assignment. This should be the same group as in the last assignment. If not, please discuss with the course staff.

Remember that the course staff is happy to help with problems you run into. For help, read all Ed posts and ask questions (that have not already been addressed), attend office hours, or meet with any course staff member either at the prearranged office hour time or at a mutually satisfactory time you arrange.

1.3 Package names

Please ensure that all Java code you submit is contained within a package (or similar, for other languages) whose name contains the NetID of at least one of your group members. Subpackages

under this package are allowed; they can be named however you would like.

1.4 Tips

Many of your optimizations will be performed at the IR level. The `--irrun` option, if implemented, should be extremely useful for verifying the correctness your optimizations.

Register allocation will be harder than it looks because you'll be doing it at the inherently more complex abstract assembly level, whereas the other optimizations will be easier to do at the IR level. Appel's chapter on register allocation will be helpful in getting the details of register allocation right, though you are not required to use exactly his algorithm.

2 Design overview document

We expect your group to submit an overview document. The Overview Document Specification outlines our expectations.

3 Building on previous programming assignments

As before, you are building upon your work from Programming Assignment 5. The protocol is the same as in prior assignments: you are required to develop and implement tests that expose any problems with your implementation, and then fix the problems. Correctness of previous assignments will count for more than it has earlier.

4 Version control

As in the last assignment, you must submit file `pa6.log` that lists the commit history from your group since your last submission.

5 Required optimizations

For this assignment, you are required to implement three optimizations:

- **Register allocation** (`reg`)
  We expect you to implement register allocation with move coalescing. You must use a live variable analysis, enabling reuse of the same register for multiple variables. You will need to handle spilling. Move coalescing will make it easier to debug other optimizations because it eliminates unnecessary temporaries. You are not required to implement Chaitin's algorithm.
- **Copy propagation** (`copy`)
- **Dead code elimination** (`dce`)
  Note that both copy propagation and dead code elimination can be done as a cascaded analysis that will save implementation effort and improve the result quality.

- **SSA (groups with CS 5120 students only)**
  Groups that include CS 5120 students are additionally required to convert their IR to SSA form.

For each optimization, think carefully about what program representation it is best done on. We suggest working through some example programs to convince yourself that you have the right analysis and program transformations worked out before doing implementation.

6 Additional optimizations

You are also required to implement at least one of the following optimizations:

- Common subexpression elimination
- Inlining function definitions
- Strength reduction with induction variables
- Loop unrolling
- Loop-invariant code motion
- Partial-redundancy elimination (this will also count for implementing CSE)
- Constant propagation along with value numbering
- A points-to analysis that makes other optimizations more effective (It may need to be context-sensitive to be effective.)

If there is some other optimization you would like to do in lieu of the optimizations on this list, consult the course staff.

7 Control-flow graphs

To be able to optimize code successfully, your compiler must be able to construct control-flow graphs for IR, and it must be able to flatten CFGs back into inline code for code generation purposes.

The compiler must also be able to generate displayable versions of CFGs in `dot` format, allowing easily visualization with Graphviz or other tools that accept this format. You will find this capability useful for debugging your optimizations, so get it working early on!

8 SSA Additions To The IR

You may implement SSA in your IR however you want, but if you are using our IR, we have updated it and the interpreter with new nodes specifically for SSA. We add nodes to support an implementation similar to Filip Pizlo's "phi/upsilon" variant of SSA. This also happens to be how CS 6120 now does SSA in its IR.

The form of SSA discussed in lecture uses φ nodes at the top of a basic block to assign different values to a variable depending on the control flow edge used to reach that basic block. A straightforward implementation makes the IR's semantics depend on the CFG, coupling the two. That isn't necesarilly a bad thing, but we didn't want to do it as we think it makes the IR more

complicated. One way of getting around this is splitting φ nodes into two: a "SET" node and a "GET" node.

```
(SET X (TEMP t0))
```

Assigns a variable "X" in some "shadow" map (different from the map of temps to values) the value of "t0".

```
(GET X)
```

Evalutes to the value assigned to "X" in the shadow map.

SET and GET can implement a statement $x = \varphi$ at the top of a basic block, b, as follows: At the bottom of every basic block with a CFG edge into b, add a SET statement assigning "x" in the shadow map the value to be assigned to the temp "x". At the top of b insert a move assigning "x" the value of (GET x). This assigns "x" a different value depending on the control flow into b, emulating a φ node.

We additionally added a new value, "UNDEF". This can be useful when a variable may not be initialized along some CFG edge into a block but it is initialized along others. Some CFG edges into that block may assign to that variable, so the top of that block may GET it from the shadow map. But, that GET can fail as its variable doesn't have a mapping in the shadow map if control flow came from a CFG edge which didn't initalize it. As a solution, the variable can be explicitly SET to UNDEF at the end of a block which doesn't initalize it, giving it a value in the shadow map.

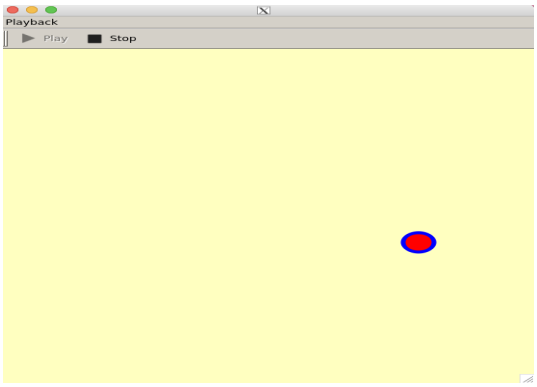
9 Rho

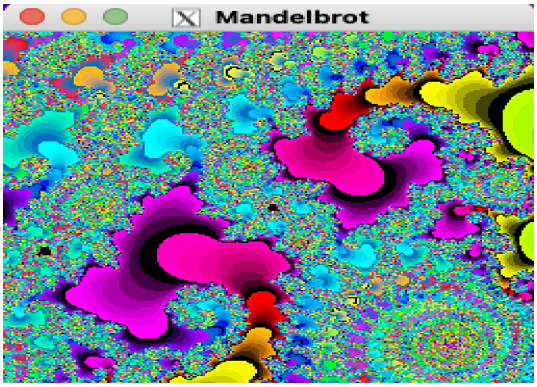
For this part of the assignment, you will extend your compiler to support the new features outlined in the Rho language specification. The language is backwards-compatible, so your compiler should accept both languages.

10 QtRho: A GUI library for Rho

To help you do interesting things with Rho programs, we are providing a basic GUI library that will let you built interactive programs with nice graphics. In particular, we provide an Rho interface to a subset of the cross-platform Qt library.

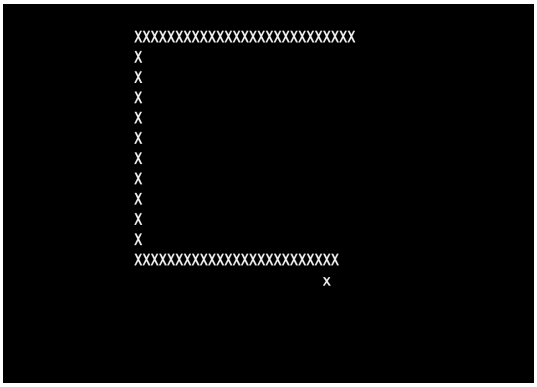
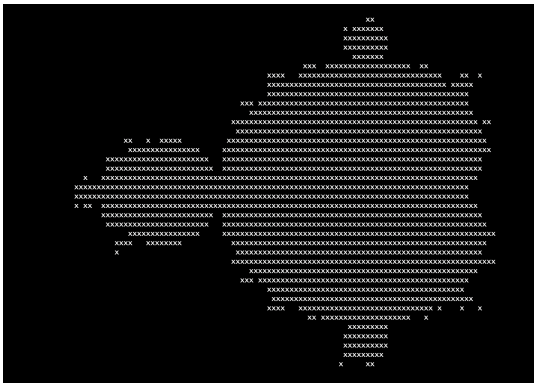
We will include some example programs using Qt for you to test your compiler. Look out for an Ed post about setup and examples. A ball animation and Mandelbrot plotter will be provided.





11 CursesRho

We also provide a text based graphics library with which you can test your compiler: curses. We will provide some example programs to supplement the programs that you may choose to write. The snake game and a Mandelbrot plotter will be provided.

12 Command-line interface

The command-line syntax is as follows:

```
etac [options] <source files>
```

Unless noted below, the expected behaviors of previously available options are as defined in the previous assignment. `etac` should support any reasonable combination of options. For this assignment, the following options are possible:

- `--help`: Print a synopsis of options.
- `--report-opts`: Output (only) a list of optimizations supported by the compiler, for example:

```
% etac --report-opts
reg
cse
copy
%
```

- `--lex`: Generate output from lexical analysis.
- `--parse`: Generate output from syntactic analysis.
- `--typecheck`: Generate output from semantic analysis.
- `--irgen`: Generate output from intermediate code generation.
  The IR is output after constant folding, if enabled, is complete.
- `--irrun`: Interpret generated intermediate code (optional).
- `--optir <phase>`: Report the intermediate code at the specified phase of optimization.
  For each source file given as `path/to/file.eta` in the command line, an output file named `path/to/file_<phase>.ir` is generated to contain the intermediate representation of the source file at the specified phase of optimization. At least the following phases must be supported:
  - `initial`: before any optimizations are performed
  - `final`: after all optimizations, if any, are complete

  You may add additional phases if you wish, but you should document them. Specifying `--optir` multiple times should generate an ir file for each phase.
- `--optcfg <phase>`: Report the control-flow graph at the specified phase of optimization.
  For each source file given as `path/to/file.eta` in the command line, and for each definition of function or procedure named `f` in the source file, an output file named

  `path/to/file_f_<phase>.dot`

  is generated to contain the control-flow graph for `f` in the `dot` format. The argument `<phase>` works in the same way as for `--optir`. Specifying `--optcfg` multiple times should generate a dot file for each phase.
- `-sourcepath <path>`: Specify where to find input source files.
- `-libpath <path>`: Specify where to find library interface files.
- `-D <path>`: Specify where to place generated diagnostic files.
- `-d <path>`: Specify where to place generated assembly output files.
- `-target <OS>`: Specify the operating system for which to generate code.
- `-O<opt>`: Enable optimization `<opt>`.
  If one of these options is used, other optimizations are off by default unless otherwise enabled. The following optimization names are standard, though your compiler probably will not implement all or even most of them:
  - `cf`: Constant folding
  - `reg`: Register allocation
  - `mc`: Move coalescing (and register allocation)
  - `cse`: Common subexpression elimination
  - `alg`: Algebraic optimizations (identities and reassociation)
  - `copy`: Copy propagation
  - `dce`: Dead code elimination
  - `inl`: Inlining
  - `sr`: Strength reduction
  - `lu`: Loop unrolling

  - `licm`: Loop-invariant code motion
  - `pre`: Partial redundancy elimination
  - `cp`: Constant propagation
  - `vn`: Local value numbering
  - `sa`: Stack-allocate non-escaping records and arrays
- `-O`: Disable all optimizations.

13 Build script

Your build script `etac-build` from previous programming assignments should remain available. The expected behaviors of the build script are as defined in the previous assignment. **The build script must be in the root directory of your submission `zip` file.** Problems within the build script from previous submissions should be fixed.

14 Benchmark test cases

Each group must submit at least 3 benchmark test cases **for each optimization** you implement. Each benchmark should conform to the standard specifications and speed up the program (it may help to write tests that repeatedly execute optimizable operations).

Each benchmark will be a valid Eta source file. A compiler `c` passes a test `t` if and only if

- `c` successfully compiles `t` into an assembly file `a`,
- assembling and linking `a` against the standard Eta library results in a runnable program `o`, and
- when executed, `o` terminates with a exit code 0 **within 3 seconds**, i.e., it terminates normally, and not as a result of assertion failing or an array out-of-bounds violation.

All test cases must

- be ASCII-encoded files,
- be valid Eta programs, according to the standard specifications (i.e., the Eta language specification and Eta type system specification),
- use only standard `io` and `conv` interfaces,
- not read input,
- contain at most 20 lines of code, excluding comments, and
- contain no line longer than 80 characters.

These test cases will also be run against the compilers of other groups, and groups will receive good karma for generating the fastest code for submitted test cases, or for submitting test cases that expose bugs in other compilers. You are welcome to develop more than 3 benchmarks for each optimization, but please choose your best 3 for the purposes of running against other compilers. All test cases submitted will be released after the assignment's due date.

We suggest you aim for benchmarks that take between 1 and 3 seconds unoptimized. Very short time intervals are hard to measure reliably; and long-running benchmarks will make your

performance testing runs too time-consuming. Where possible, your benchmarks should also produce verifiable results, and ideally, self-verify.

15 Submission

Before submitting, you should first prepare your repository. Make sure that your repository is complete and ready to be graded. In particular, it should include all of the following:

- All source code needed to compile and run your project, including any required third-party libraries.
- All test cases and any scripts used to run them.
- A design overview document, as described in §2, placed in the root of your repository.
- A `README` file explaining how to navigate, build, run, and test your code (this may overlap with the design overview).
- Benchmark test cases: three benchmark test cases per optimization to be run against other compilers. **These test cases must reside in directory "benchmarks" at the root of your repository.**
- No unnecessary files, such as large binaries, generated files, IDE metadata, or OS metadata.

Once your repository is ready, go to your repository on the Cornell GitHub website and create a new release (found in the right sidebar). Use the programming assignment title—e.g., "Programming Assignment 6: Optimization" as the title of your release, and set the tag to `submissions`. Publish your release and download the release ZIP file. Finally, submit the release ZIP file on CMSX before the deadline of Monday, May 11, 4:30PM. **All of these steps must be completed on time.**

Your repository should be self-contained, meaning that graders should be able to clone it and build your project without additional setup. Using Git submodules is allowed; if you do so, ensure that `git clone --recursive your_repo` retrieves everything needed to build and run your project.

For smaller libraries, it is often easy and effective to include the source code directly, but be sure to make clear what is library code, e.g. by package name. JAR files also work well, but sometimes do not include Javadoc and are less well integrated with your IDE. Your mileage may vary.

If you use a lexer or parser generator, please include the input files, e.g., "*.flex". Your "etac-build" should use these files to generate source code, and you should not submit the corresponding generated source code file (e.g. "*.java").

Finally, remember that clear and well-written documentation—including commit messages, the design overview, and the `README`—does not just help your graders; they also help you and your partners.

16 AI Policy

You may use AI tools such as ChatGPT to assist with small, well-scoped pieces of code. However, you should not use AI tools to generate large portions of your code or core functionality. Using AI too much is a common pitfall and tends to backfire in this course.

In your design overview document, please write down where and how you used AI tools, if at all.

The Vibe Mirage.

Who needs to write code anymore—don't we just ask generative AI to do the work for us? This approach may sound like it will save time and effort, but it rarely does. AI-generated code often looks convincing until it is tested carefully and reveals subtle bugs that cost far more time than they save. Because each stage of the project builds on your existing code, your foundations must be solid. Relying on AI can undermine your understanding of your own code, and that understanding is critical for success.