CS4120/4121/5120/5121—Spring 2016
Programming Assignment 6
Optimization
Due: Wednesday, April 27, 11:59PM

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. 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

- None yet; watch this space.

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.

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 Piazza 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 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.

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.

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, there are three optimizations that you are required to implement:

- **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.
- **Unreachable code elimination** (`uce`)
  We expect unreachable code to be eliminated. This should work in conjunction with constant propagation, if implemented, and constant folding (`cf`). Conditional constant propagation is one option to do this.
- **Common subexpression elimination** (`cse`)
  This is a global optimization, so it will require an *available expressions* analysis.

For each optimization, you should 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.

- Copy propagation along with dead code 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 Command-line interface

The command-line syntax is as follows:

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

Unless noted below, the expected behaviors of previously available options are as defined in the previous assignment. `xic` 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:

```
% xic --report-opts
reg
uce
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.xi` 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.xi` 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
  - `uce`: Unreachable code elimination
  - `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
- `-O`: Disable all optimizations.

This option is redundant if one of the `-O<opt>` options is used.

- `-O-no-<opt>`: Disable only optimization `<opt>`.
  Other optimizations are on by default unless otherwise disabled.

9 Build script

Your build script `xic-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.

10 Benchmark test cases

Each group must submit at least 3 benchmark test cases **for each optimization**. 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 Xi 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 Xi 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 Xi programs, according to the standard specifications (i.e., the Xi language specification and Xi 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.

11 Submission

You should submit these items on CMS:

- `overview.txt/pdf`: Your overview document for the assignment. This file should contain your names, your NetIDs, all known issues you have with your implementation, and the names of anyone you have discussed the homework with. It should also include descriptions of any extensions you implemented.
- A `zip` file containing these items:
  - *Source code*: You should include all source code required to compile and run the project. Please ensure that the directory structure of your source files is maintained within the archive so that your code can be compiled upon extraction. If your code depends on any third-party libraries, please include compilation instructions in your overview document.
    Include your parser generator input file, e.g., `*.cup`, as well as the generated code. If you use a lexer generator, please include the lexer input file, e.g., `*.flex`, as well as the generated code.
  - *Tests*: You should include all your test cases and test code that you used to test your program. Be sure to mention where these files are and to describe your testing strategy in your overview document.
  - *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 the `zip` file directory hierarchy.**

  Do not include any non-source files or directories such as `.class`, `.classpath`, `.project`, `.git`, and `.gitignore`.
- `pa6.log`: A dump of your commit log since your last submission from the version control system of your choice.