CSE 131: Compilers, Project 1, Spring 2024

Assigned: April 4, 2024, 10% of course grade
(100 points total)
Due on Gradescope by 11:59pm on April 25th, 2024

1 Project Description

In this semester, through 3 projects, we will build a compiler for a small pedagogic programming language named Tiger, targeting the MIPS32 architecture, through an intermediate representation called Tiger-IR. The three projects will be:

- Project 1 (middle end): Static analysis and code optimization of Tiger-IR
- Project 2 (back end): MIPS32 code generation from Tiger-IR
- Project 3
  - Option 1 (front end): Generate a parser for Tiger source code
  - Option 2 (program analysis): Control-flow analysis based on LLVM

Each of these projects will have a file input and output interface, to enable them to be implemented independently. You may choose any of the following implementation languages: Java, Python, C, or C++. You must provide complete self-contained instructions in your submission on how to build and run your project (including version numbers of programming system and support libraries), as well as the `build.sh` and `run.sh` scripts to do so. We've provided helper code in Java which may make it more convenient for teams that use Java as their implementation language. Teams that use other languages may need to port some/all of our Java helper code to their implementation language.

In this first project, you will build a simple optimizer (middle end) for Tiger-IR. The reference manual for Tiger-IR can be found in the course project repository mentioned in Section 2. Your optimizer must be in one of the following languages: Java, Python, C, or C++. The optimizer should read in a text-formatted Tiger-IR program, perform optimizations on it, and write the optimized IR code to a file.

The goal of your optimizer is to improve the performance of a given program. In this project, we measure the performance of an IR program by collecting the *dynamic instruction count*, which is the number of instructions executed by an IR interpreter during an execution. Please note that this is a different goal from minimizing the number of instructions in a program, which is called the

static instruction count, as sometimes it is possible to insert more static instructions to obtain a reduced dynamic instruction count. As indicated below, we identify a reference optimization (dead code elimination using reaching definitions) that suffices to obtain full credit. However, there is no restriction on which optimizations your optimizer should contain, as long as they are correct, and meet the dynamic instruction count performance goals outlined below in Section 1.2.

The deliverables for the project are a design document and code submission, which are described below in Sections 1.1 and 1.2.

1.1 Design of your Optimizer (30 points)

With your submission, please include a design document called 'design.pdf' in the zip file mentioned in Section 1.2. This document should briefly describe the following:

1. High-level architecture of your optimizer, including the analysis and optimization algorithm(s) implemented, and why you chose that approach.
2. Low-level design decisions you made in selection of implementation language, and their rationale.
3. Software engineering challenges and issues that arose and how you resolved them.
4. Any known outstanding bugs or deficiencies that you were unable to resolve before the project submission.
5. Build and usage instructions for your optimizer.
6. A summary of the test results for the public test cases (see Section 1.2).

1.2 Evaluation (70 points)

The grading on the performance of your optimizer's output is based on test cases. A test case is defined as a <test-IR-program, test-input> pair. First we will execute your `build.sh` script to build your optimizer executable. Then we will run your optimizer on a set of test programs using your `run.sh` script. 20 points will be allocated for *public* test cases, which we will provide you, and 50 points will be allocated for *hidden* test cases, which we will use when grading your project. For each test case, we will perform two steps to evaluate your optimizer:

1. Execute your `run.sh` script to run your optimizer on a test-IR-program to obtain an optimized IR program.
2. Run the optimized IR programs on an IR interpreter (which we will provide you for testing purposes) with test-input, and report the dynamic instruction count obtained during the execution. (Label instructions will not be included in the dynamic instruction count.)

In any of the following cases, you will get 0% of the score for a test case:

- Your optimizer crashes, or produces invalid code.
- The optimized program does not produce the desired output when being interpreted.

- The optimized program produces the desired output, but its dynamic instruction count is $\geq$ that of the original IR program.

To get 50% of the score for a test case, the IR code generated by your optimizer will need to have the same or better performance compared with an optimizer (optimizer A) that does the following optimization:

- Simple dead (useless) code elimination without branch removal
  - Based on the algorithm in Lecture slides (without reaching definitions)

To get 100% of the score for a test case, the IR code generated by your optimizer will need to have the same or better performance compared with an optimizer (optimizer B) that does the following optimization:

- Dead (useless) code elimination without branch removal
  - Based on the algorithm in Lecture slides (with reaching definitions)

The optimizations mentioned above are what we expect most project teams to implement. However, you are welcome to implement other/additional optimizations that you choose. Your goal is to make your optimizer generate equivalent or better code than optimizer A or B does, in terms of performance (dynamic instruction count). Assuming successful completion. The actual score that you receive for a test case will be an interpolation between the 50% and 100% reference points mentioned above.

2 Provided Code

We have provided some Java helper code and the IR interpreter in:
Google Drive: `https://drive.google.com/drive/folders/1NF2Oo-DdqVHjLXijViiHnAMPsYkVImzt?usp=drive_link`

2.1 Helper Code

The Java helper code is located under `/Project-1/materials/src/ir`. It defines some data structures to represent the IR in memory, and provides functionality for reading and printing the IR.

If you decide to use the helper code in your project, please be sure to carefully read all the source files, except `IRReader.java` and `IRPrinter.java`. For these two files, you only need to know how to use the APIs they provide, which is demonstrated in `/Project-1/materials/src/Demo.java`, but you are welcome to read the two files also. The `Demo.java` file also includes some important documentation and usage patterns of the helper code.

It is not required to use the helper code, and you are welcome to modify the helper code as you see fit.

2.2 Tiger-IR Interpreter

The source code of the IR interpreter is in `/Project-1/materials/src/IRInterpreter.java`. It also uses the helper code. Please refer to `/Project-1/materials/README.md` for more information about how to build and use it. We will grade your project with the same IR interpreter, but you can feel free to modify it for debugging purposes as needed.

3 Submission

On Gradescope, submit a single ZIP file that contains:

- The complete source code of your project, as described in Section 1.2.
- A `build.sh` script in the top level directory which builds your project.
- A `run.sh` script in the top level directory which runs your project executable on an input path to an ir file `path/to/file.ir` and outputs an `out.ir` file.
- The design.pdf file described in Section 1.2.
- You can submit as many times as you want; we will use your active submission for the final grade.

4 Collaboration

We will award identical grades to each member of a given project team, unless members of the team directly register a formal complaint. We assume that the work submitted by each team is their work solely. Any clarification question about the project handout should be posted on the course's public Piazza message board. Any non-obvious discussion or questions about design and implementation should be either posted on the course's Piazza message boards privately for the instructors or presented in person during office hours. If the instructors determine that parts of the discussion are appropriate for the entire class, then they will forward selections. Under no condition is it acceptable to use code written by another team, or obtained from any other source. As part of the standard grading process, each submitted solution will automatically be checked for similarity with other submitted solutions and with other known implementations.