**CMSC 22600** **Compilers for Computer Languages** **Project 3**
**Autumn 2016** **November 29, 2016**

# Optimization (extra credit)

**Due: December 6, 2016**

## 1 Introduction

For extra credit, you may add a SOIR to SOIR optimization pass to your compiler. We will update the project repositories with a placeholder for the optimization pass (in `soir-opt`) and a compiler flag for enabling optimization ("`-O`"). If you choose to implement optimizations for SOIR, please add a file named `OPTIMIZATION` to the root of your source tree and include a description of what optimizations you have implemented.

## 2 Possible optimizations

The class of optimizations that we consider are all examples of *contractions* (or *shrinking*) transformations of the code. This class of optimizations tend to provide the most "bang for the buck" as they are usually straightforward to implement and almost always result in better performance. To support these optimizations, you will need information about how local variables are defined and about how many uses they have.

Note that this list is not meant to be exhaustive.

### 2.1 Compile-time arithmetic

When one or more arguments to an arithmetic operator (or relational operator) are known at compile time, it is possible to replace arithmetic expressions with constants and/or simpler computations.

### 2.2 Dead-variable elimination

When the use count of a variable is zero, and the variable is not bound to a side-effecting expression, then it is possible to delete the variable (and the expression that defines it). This optimization is useful as a cleanup for other transformations. It also tends to cascade, since the use counts of the left-hand-side variables will be decreased.

### 2.3 Compile-time conditional evaluation

If we have a conditional (**`if`** or **`exit_if`**) that tests a known boolean value, then we can evaluate it at compile time. In the case of **`exit_if`**, a **`true`** value cause the loop to be flattened away leaving just the header code and a **`false`** value creates an infinite loop. Note that if your code generator expects all loops to include an **`exit_if`**, then you may not be able to eliminate it in the **`false`** case.

### 2.4 Compile-time data-structure accessing

Similar to compile-time arithmetic, it is possible to index into tuples, class-metadata tables, and index tables at compile time. When accessing class metadata and index tables, however, you must be careful to avoid breaking subtyping and dynamic method dispatch.

## 3 Submission

This extra-credit work is due at the same time as Project 4 (**11pm** on **Tuesday December 6**) and will be collected as part of Project 4. Please remember to include the `OPTIMIZATION` file in your repository (do not forget to do an `svn add`!).

## 4 Document history

**November 29, 2016** Original version.