# 1 Introduction

Compilers translate programs written in a source language (e.g., Java) into semantically equivalent programs in some target language (e.g., assembly code). They let us make our source language arbitrarily abstract so we can write programs in ways that humans understand while letting the computer execute programs in ways that machines understand. In a perfect world, such translation would be straightforward. Reality, however, is unforgiving. Straightforward compilation results in clunky target code that performs a lot of redundant computations. To produce efficient code, we must rely on less-than-straightforward methods. Typical compilers go through a stage of *optimization*, whereby a number of semantics-preserving transformations are applied to an *intermediate representation* of the source code. These then (hopefully) produce a more efficient version of said representation. Optimizers tend to work in *phases*, applying specific transformations during any given phase.

Global value numbering (GVN) is such an analysis performed by many highly-optimizing compilers. Its roots run deep through both the theoretical and the practical. Using the results of this analysis, the compiler can identify expressions in the source code that produce the same value—not just by lexical comparison (i.e., variables having the same name), but by proving equivalences between what's actually computed at runtime. These expressions can then be simplified by further algorithms for redundancy elimination. This is the very essence of most compiler optimizations: avoid redundant computation, giving us code that runs as quickly as possible while still following what the programmer originally wrote.

High-level, dynamic languages tend to suffer from efficiency issues: they're often interpreted rather than compiled, and perform no heavy optimization of the source code. However, the Factor language (`http://factorcode.org`) fills an intriguing design niche, as it's very high-level yet still fully compiled. It's still young, though, so its compiler craves all the improvements it can get. In particular, while Factor currently has a *local* value numbering analysis, it is inferior to GVN in several significant ways.

In this thesis, we explore the implementation and use of GVN in improving the strength of optimizations in Factor. After establishing some preliminary terminology and concepts in **??**, we turn our attention to the details. Because Factor is a young and relatively unknown language, **??** provides an short tutorial, laying a foundation for understanding the changes. **??** describes the overall architecture of the Factor compiler, highlighting where the exact contributions of this thesis fit in. Finally, **??** goes into detail about the existing and new value numbering passes, closing with a look at the results achieved and directions for future work. In total, the following Factor libraries were written for this thesis:

- A `graphviz` library, which provides bindings to create and manipulate graphs in Factor and output them using Graphviz (`http://graphviz.org`).
- The `compiler.cfg.graphviz` library, which sues the Graphviz bindings to output images of Factor's low-level intermediate representation. This is responsible for the illustrations of optimization passes seen in **??**.
- The `compiler.cfg.gvn` module, which was created by copying the existing value numbering code (from `compiler.cfg.value-numbering`) to make the pass global.

All code was written atop Factor version 0.94, and copies of it can be found in the appendices. In the unlikely event that you want to cite this thesis, you may use the following BibTeX entry:

```
@mastersthesis{vondrak:11,
  author = {Alex Vondrak},
  title  = {Global Value Numbering in Factor},
  school = {California Polytechnic State University, Pomona},
  month  = sep,
  year   = {2011},
}
```