Franz Mandl, BSc

Creating a Web Application to Visualize Compiler Optimizations

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisors

Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa
Dipl.-Ing. Dipl.-Ing. Dr.techn. Roxane Koitz-Hristov, BSc

Institute of Software Technology
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa

Graz, September 2022

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Abstract

Compilers are applications that translate code written in one language into another, usually without a graphical user interface. Programmers are often left in the dark about the internal workings of a compiler, such as code optimization decisions. To address such issues, a research area emerged around visualization applications for educational purposes. Students indicate that they are more confident in the knowledge they have gained from using such applications compared to using only written material.

In the scope of this thesis, we developed a web application to visualize compiler optimizations called Visual Optimizer (VisOpt). We started by analyzing similar, already existing applications in the field of compiler visualization for computer science education. Based on the newly acquired knowledge, we implemented VisOpt and released it as an open-source project.

To measure the impact of VisOpt on student learning, we conducted a controlled experiment using a pre-test-post-test design. Students of the course “Compiler Construction” at Graz University of Technology could participate in the experiment voluntarily. They were randomly divided into an experimental group working with VisOpt and a control group.

The experiment led to the conclusion that using VisOpt in combination with written material has the same effect on learning as using only written material. However, students using VisOpt reported to be slightly more confident about their level of competence and fully agree that using VisOpt has improved their understanding of code optimization.

Contents

1. Introduction

Many researchers believe that visualization applications help to understand the internal workings of algorithms or programs. The research field surrounding the use of visualization in computer science education began in 1981 with the premier of the movie *Sorting Out Sorting* created by Baecker and Sherman (1981). Since then, many visualization applications have been developed, although some empirical research concludes that the use of visualization as a teaching aid does not significantly improve student understanding (Binkley et al., 1998; P. R. Osztián, Kátai, and E. Osztián, 2020; Stasko, Badre, and Lewis, 1993).

Nevertheless, the use of visualization applications in computer science education increased over the years (Fouh, Akbar, and Shaffer, 2012). Some applications visualize the internal workings of compilers, which serve as the basis for this thesis. For example, Boyd and Whalley (1993) developed an interactive visualization application that can be used as a teaching aid for compiler courses. The idea was that students would learn for themselves how a program gets optimized by showing the changes made to a program during an optimization phase.

Five years later, a similar application was published by Binkley et al. (1998), who also conducted an experiment with their application. The experiment found that students' understanding of code optimization did not improve significantly after using their application. Nevertheless, students who learned with their application reported having more confidence in their newly acquired knowledge than students who learned with written material only.

Over the next few years, web browsers became more sophisticated, allowing complex applications to run directly in the browser without the need to install additional software. Such a web application was developed by Devkota et al. (2020). It visualizes the correlation between the original source code and the optimized machine code. By navigating through different representation methods in multiple views, users can explore this correlation for themselves.

The main goal of this thesis is the development of a new web application in the field of compiler visualization for educational purposes called VisOpt. It is based on several open-source web frameworks that allow us to create a complex web application in a structured way, such as React (Meta Platforms Inc., 2013) and Spring (VMware Inc., 2002). In addition to these frameworks, our application contains a compiler based on ANother Tool for Language Recognition (ANTLR) (Parr, 1992).

The second goal of this thesis is to measure the impact of using VisOpt while learning code optimization. Therefore, we conducted an experiment to determine whether using the application during learning significantly improves students' understanding of code optimization.

1.1. Requirements

The resulting application is primarily intended to be used as teaching aid in the course "Compiler Construction" at Graz University of Technology. Therefore, several requirements were defined in advance:

- The resulting application should be web-based and not require the user to install any additional software.
- The application should take a program written in a Java-like language called Jova as input. A grammar definition of Jova already exists at Graz University of Technology and is used by the application.
- For the visualization, a flow graph as well as an assembly language for Java bytecode shall be used.
- The user should be able to enable and disable individual optimization techniques.
- The user should also be able to navigate manually and automatically through the single optimization steps transforming the program.

1.2. Outline

The remainder of this thesis is structured as the research was conducted.

In Chapter 2 on page 5, we summarize the theoretical background of code optimization that is necessary to understand and implement code optimization for our application.

Next, we searched for related work in the field of compiler visualization for computer science education, as described in Chapter 3 on page 17. Therefore, we conducted an online search using the keywords `compiler`, `optimization`, `animation`, and `visualization`. If there was an application described in the found material, we tried to execute it. During this process, we gathered useful information about how these applications work and what capabilities they offer.

Then, we implemented our own application based on the newly acquired knowledge using state-of-the-art web technologies, as described in Chapter 4 on page 23.

In Chapter 5 on page 39, we describe an experiment to analyze whether the newly implemented application is useful in computer science education.

Finally, we summarize the results of this thesis in Chapter 6 on page 47.

2. Theoretical Background

In this chapter we start with a general description of the structure of a compiler. Then we give an overview of optimization techniques and explain when they are performed during compile time.

2.1. Compiler

A compiler is an application that translates code written in a source language into code written in some target language. Typically, a compiler is split into an analysis part and a synthesis part (Aho et al., 2006).

Each part consists of phases in which the program is transformed from one representation to another (Aho et al., 2006). The source code is basically a character stream, which is also the first representation we encounter. The lexical analysis phase takes the character stream as input and separates it into tokens such as constants, keywords, operators, and identifiers. The resulting token stream is passed as input to the syntax analysis phase, which outputs a syntax tree. This syntax tree is then checked for type errors using a symbol table in our semantic analysis phase, which is also the last phase of our analysis part (Tremblay and Sorenson, 1985).

The next phase belongs to the synthesis part and uses the syntax tree and the symbol table to generate an intermediate representation. To represent our code, we can use three-address code. In three-address code, an instruction has at most three operands.

In an optional optimization phase, the intermediate representation gets transformed into a more optimized one. The resulting intermediate representation is transformed into target code in the code generator phase.

Finally, the target code gets optimized again in a second optional optimization phase, resulting in an even more optimized target code.

2.2. Code Optimization

The goal of code optimization is to generate improved code that performs better on certain objectives. For example, some objectives might be code size, execution speed, or power consumption (Aho et al., 2006). The behavior of the program must be preserved during code optimization.

Code optimization distinguishes between machine-independent and machine-dependent optimization as well as local and global optimization.

2.2.1. Machine-Independent and Machine-Dependent Optimization

As the name implies, machine-independent optimization performs code improvements without knowing which machine the code is running on. Both the input and the output of a machine-independent optimization phase is an intermediate representation. Machine-dependent optimization, on the other hand, takes the machine type into account, resulting in optimized target code.

In the scope of this thesis, we will focus only on machine-independent optimization.

2.2.2. Local and Global Optimization

Local optimization improves code within a maximum sequence of consecutive instructions, where the flow of control can only

(a) enter the sequence through the first instruction of the sequence and
(b) leave only through the last instruction of the sequence.

Such a sequence of instructions is called a basic block. Only the last instruction in a basic block is allowed to be a halting, branching, or jumping instruction. Peephole optimizations are simple but effective local optimization techniques that replace instructions within a small sequence of consecutive instructions, also called a *peephole*, by only looking at the peephole (Aho et al., 2006).

While local optimization only takes into account what happens inside a basic block, global optimization on the other hand considers what happens across basic blocks, e.g., by using a control flow graph.

2.2.3. Control Flow Graph

A control flow graph is a directed graph in which nodes represent basic blocks and edges represent possible transfers of control between these basic blocks (Shivers, 1991). To indicate the flow of control's start and end, we add two special nodes labeled ENTRY and EXIT.

The corresponding control flow graph of Listing 2.1 on the next page is illustrated in Figure 2.1 on the following page using a node-link representation.

2.2.4. Algebraic Simplification

Algebraic simplification is an optimization technique that uses algebraic laws to simplify code. For example, we use algebraic identities, such as

```
x = x + 0          x = 0 + x
x = x - 0
x = x * 1          x = 1 * x
x = x / 1
```

to improve our code. We can use a peephole optimizer to search for such instructions and eliminate them in the peephole, since the value of x remains the same before and after such an instruction (Aho et al., 2006).

```
int a, b;
a = 50;
b = 75;
if (a < b) {
     if (a != 0) {
          print("true");
     }
} else {
     print("false");
}
a = a + 1;
while (a > b) {
     print("while");
}
return 0;
```

Listing 2.1: Example code written in Jova

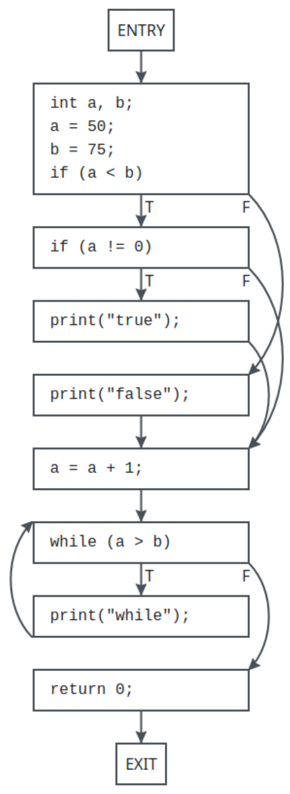


Figure 2.1.: Node-link representation of the control flow graph of Listing 2.1

2.2.5. Common Subexpression Elimination

By eliminating common subexpressions, we can avoid unnecessary computations that result in the same values (Aho et al., 2006). We introduce temporary variables to store the results of these computations. For example, applying this technique on Listing 2.2 results in Listing 2.3 .

```
a = 3 * 4 / 2;
b = 3 * 4 * 2;
result = (a + 10) * (a + 10);
```

Listing 2.2: Example code before common subexpression elimination

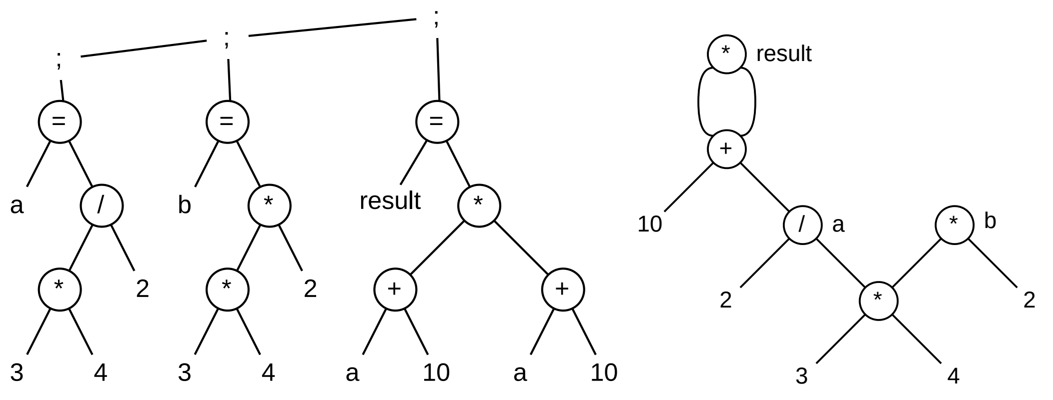
```
tmp1 = 3 * 4;
a = tmp1 / 2;
b = tmp1 * 2;
tmp2 = a + 10;
result = tmp2 * tmp2;
```

Listing 2.3: Example code after common subexpression elimination

To identify common subexpressions, we need to transform a basic block into a Directed Acyclic Graph (DAG). Similar to a syntax tree, the leaves in a DAG correspond to atomic operands while interior nodes correspond to operators (Aho et al., 2006). While creating a DAG, we need to keep track of which branches compute the same values and link them accordingly. A node in a DAG has more than one parent if the node is a common subexpression. Figure 2.2 on the next page shows an example of a syntax tree and the corresponding DAG.

2.2.6. Constant Folding

Folding constants means that constant expressions are evaluated at compile time and replaced with their constant value (Aho et al., 2006). For example, we may replace the expression $2 * 3 + 4$ with 10. We can use a peephole optimizer to search for such expressions and replace them in the peephole.



(a) Syntax tree (b) DAG

Figure 2.2.: Syntax tree and DAG for basic block in Listing 2.2 on the previous page

Since we are not allowed to change the behavior of the program, we must be careful about overflows and underflows. We need to evaluate them at compile time just as they are evaluated at runtime. Also, we cannot fold expressions that would raise an error at compile time, like 1 / 0, and leave them unchanged in the program.

2.2.7. Constant Propagation

We can replace variable references by constant values if that variable always has the same constant value at this point in the code (Aho et al., 2006). For example, in Listing 2.4 we may replace variable `i` with the value 2, resulting in Listing 2.5 .

```
i = 2;
result = i * i + i;
```

Listing 2.4: Example code before constant propagation

```
i = 2;
result = 2 * 2 + 2;
```

Listing 2.5: Example code after constant propagation

Constant propagation is a local optimization and is therefore applied to a basic block. While iterating over all instructions in a basic block, we associate a variable with a value when we encounter an assignment of a constant value. Further usages of this variable are then replaced by the constant value until it is reassigned.

2.2.8. Copy Propagation

The idea behind copy propagation is that after an assignment (or copy) statement `u = v`, we may use `v` over `u` afterwards (Aho et al., 2006). For example, using variable `i` over `copy` in Listing 2.6 results in Listing 2.7 .

```
copy = i;
result = copy * copy + copy;
```

Listing 2.6: Example code before copy propagation

```
copy = i;
result = i * i + i;
```

Listing 2.7: Example code after copy propagation

Copy propagation is like constant propagation, except that we associate a variable with another variable when we encounter a corresponding assignment.

2.2.9. Dead Code Elimination

The elimination of instructions that compute values that are never used is called *dead code elimination* (Aho et al., 2006). A variable is *dead* at a point in a program if the value of a variable is not accessed before the next time it gets assigned. We may eliminate assignments to dead variables. If the value of a variable may get accessed later in another basic block, we call it *live on exit*.

Applying dead code elimination on Listing 2.8 with variable `b` live on exit eliminates the first assignment of `b` as well as the assignment of `c` as shown in Listing 2.9 .

```
a = 201;
b = 302;
c = a * 2;
b = a + 3;
```

Listing 2.8: Example code before dead code elimination

```
a = 201;
b = a + 3;
```

Listing 2.9: Example code after dead code elimination

Live-Variable Analysis

To determine whether a variable is live or dead on exit, we analyze the flow of data along execution paths. Since we cannot keep track of all program states along all possible paths, we abstract out certain details we need for our analysis (Aho et al., 2006).

In live-variable analysis, we are interested in the assignments and usages of variables in a basic block. Therefore, we associate four sets to each basic block B:

- def_B contains variables that get assigned in B before any usage of that variable in B (Aho et al., 2006).
- use_B contains variables whose values may be used in B before any assignment of that variable in B (Aho et al., 2006).
- IN[B] contains live on entry variables of B.
- OUT[B] contains live on exit variables of B.

To compute the IN and OUT sets for each basic block, the algorithm shown in Figure 2.3 on the facing page can be used. The algorithm takes the control flow graph with the def and use sets already computed for each basic

```
1: for each basic block $B$ do
2:     $\text{IN}[B] = \emptyset$
3: end for
4: while changes to any IN occur do
5:     for each basic block $B$ other than EXIT do
6:         $\text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S]$
7:         $\text{IN}[B] = use_B \cup (\text{OUT}[B] \setminus def_B)$
8:     end for
9: end while
```

Figure 2.3.: Iterative algorithm to compute live variables (Aho et al., 2006)

block as input. After the IN and OUT sets have converged, the OUT sets contain the live on exit variables for each basic block. Since we only add variables to the IN and OUT sets and never remove variables, they converge eventually.

This algorithm is also called *flow propagation algorithm*, since information is propagated in reverse order along the edges of the control flow graph. In our case, the flow information is a finite set of the variables, so the algorithm converges in polynomial time (Tonella and Potrich, 2005).

Unreachable Code Elimination

Unreachable code is code that never gets executed at runtime under any circumstances (Debray et al., 2000). Therefore, we can eliminate it during compile time. For example, in Listing 2.10 we may eliminate the else branch by replacing the entire if-then-else statement with the print statement in the then branch.

```
if (true) {
    print("then branch");
} else {
    print("unreachable else branch");
}
```

Listing 2.10: Example code before unreachable code elimination

2.2.10. Reduction in Strength

Replacing more expensive operators with cheaper ones is called *reduction in strength* (Aho et al., 2006). For example, a multiplication or division by a power of two can be replaced by a cheaper addition or bit shift operation, such as:

```
x * 2 = x + x          2 * x = x + x
x * 4 = x « 2          4 * x = x « 2
x / 4 = x » 2
```

We can use a peephole optimizer to search for such instructions and replace them in the peephole with cheaper instructions.

2.2.11. Execution Order of Optimizations

We have discussed several optimization techniques so far, but we need to combine them to optimize a program. Therefore, we need to decide in which order to apply them.

By starting with copy propagation, we replace variables with other variables, thus reducing the number of different variable usages. By applying constant propagation next, we might reduce the total amount of variable usages. Now that there are fewer variable usages and more constant values, this is a good opportunity to apply constant folding. As more complex constant expressions have been evaluated, we next perform algebraic simplification and reduction in strength. Before eliminating common subexpressions, we first eliminate dead code, since we do not want to introduce unnecessary temporary variables. Last, we apply common subexpression elimination.

In conclusion, we used the following order of applying optimizations on a program:

1. Copy Propagation
2. Constant Propagation
3. Constant Folding
4. Algebraic Simplification
5. Reduction In Strength

6. Dead Code Elimination
7. Common Subexpression Elimination

We apply this sequence to each basic block one after the other. We repeat this until there is nothing left to optimize and the intermediate representation of the program does not change.

3. Related Work

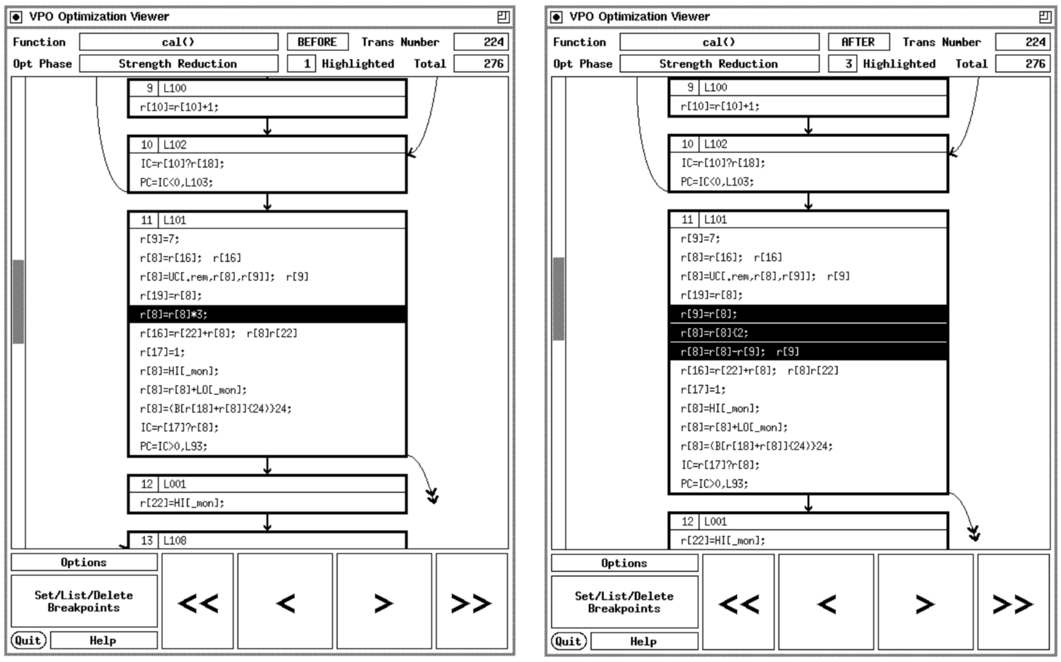
There exists a considerable amount of related work in the field of compiler visualization. They often describe applications specially designed for computer science education. In this chapter, we summarize the key findings from the related work that most influenced this work.

3.1. XVPODB

The X-windows Very Portable Optimizer DeBugger (XVPODB) is a graphical optimization viewer developed by Boyd and Whalley (1993). It visualizes optimizations performed by the Very Portable Optimizer (VPO) developed by Benitez and Davidson (1988). VPO and XVPODB run in separate processes, with VPO running in debug mode, allowing XVPODB to control execution and access internal data structures via Unix sockets (Boyd and Whalley, 1993).

VPO uses Register Transfer Lists (RTLs) as intermediate representation, which is also used for visualization by XVPODB. For example, the RTL of a register-to-register integer addition is `r[1] = r[1] + r[2]` (Benitez and Davidson, 1988).

VPO uses different optimization phases to transform the program into a semantically equivalent but optimized program (Boyd and Whalley, 1993). For example, one such optimization phase is common subexpression elimination. In general, an optimization phase consists of one or multiple smaller transformations. Such smaller transformations include adding, replacing, or removing one or more lines of code.



Function cal() BEFORE Trans Number 224
Opt Phase Strength Reduction 1 Highlighted Total 276

```
9  L100
r[10]=r[10]+1;

10 L102
IC=r[10]?r[18];
PC=IC<0,L103;

11 L101
r[9]=7;
r[8]=r[16];  r[16]
r[8]=UC[.rem,r[8],r[9]];  r[9]
r[19]=r[8];
r[8]=r[8]*3;
r[16]=r[22]+r[8];  r[8]r[22]
r[17]=1;
r[8]=HI[_mon];
r[8]=r[8]+LO[_mon];
r[8]=(B[r[18]+r[8]]{24)}24;
IC=r[17]?r[8];
PC=IC>0,L93;

12 L001
r[22]=HI[_mon];

13 L108
```

(a) Before state

Function cal() AFTER Trans Number 224
Opt Phase Strength Reduction 3 Highlighted Total 276

```
9  L100
r[10]=r[10]+1;

10 L102
IC=r[10]?r[18];
PC=IC<0,L103;

11 L101
r[9]=7;
r[8]=r[16];  r[16]
r[8]=UC[.rem,r[8],r[9]];  r[9]
r[19]=r[8];
r[9]=r[8];
r[8]=r[8]{2;
r[8]=r[8]-r[9];  r[9]
r[16]=r[22]+r[8];  r[8]r[22]
r[17]=1;
r[8]=HI[_mon];
r[8]=r[8]+LO[_mon];
r[8]=(B[r[18]+r[8]]{24)}24;
IC=r[17]?r[8];
PC=IC>0,L93;

12 L001
r[22]=HI[_mon];
```

(b) After state

Figure 3.1.: Screenshot of XVPODB's main window during a reduction in strength (Boyd and Whalley, 1993)

XVPODB visualizes them by showing the program before and after such a transformation. The main window of XVPODB is shown in Figure 3.1 and is divided into three sections.

The top section shows which method is currently being optimized, which optimization phase caused the current transformation, whether we are before or after the current transformation, how many lines of code are currently affected, the number of the current transformation, and how many transformations there are in total.

In the middle section, the user can see the control flow graph as node-link representation, where each node contains RTL code of the corresponding basic block. Lines of code that are changed by the current transformation are highlighted. Highlighted lines before a transformation get removed, while highlighted lines after a transformation were either inserted or altered (Boyd and Whalley, 1993).

The bottom section allows the user to control the visualization via buttons. Clicking the step forward button (>) takes us from the before state of a transformation (Figure 3.1a on the preceding page) to its after state (Figure 3.1b on the facing page).

Clicking the same button again changes from the after state to the before state of the next transformation and so on. The step backward button (<) leads in the opposite direction. XVPODB also allows the user to set breakpoints when a certain event occurs, e.g., the current transformation has a certain number. Clicking the continue forward (>>) or continue backward (<<) button continues the optimizer until the start, end, or a breakpoint is reached.

Unfortunately, we were unable to run XVPODB ourselves because the official download link (Florida State University, 1993) is no longer reachable and no other copies could be found elsewhere.

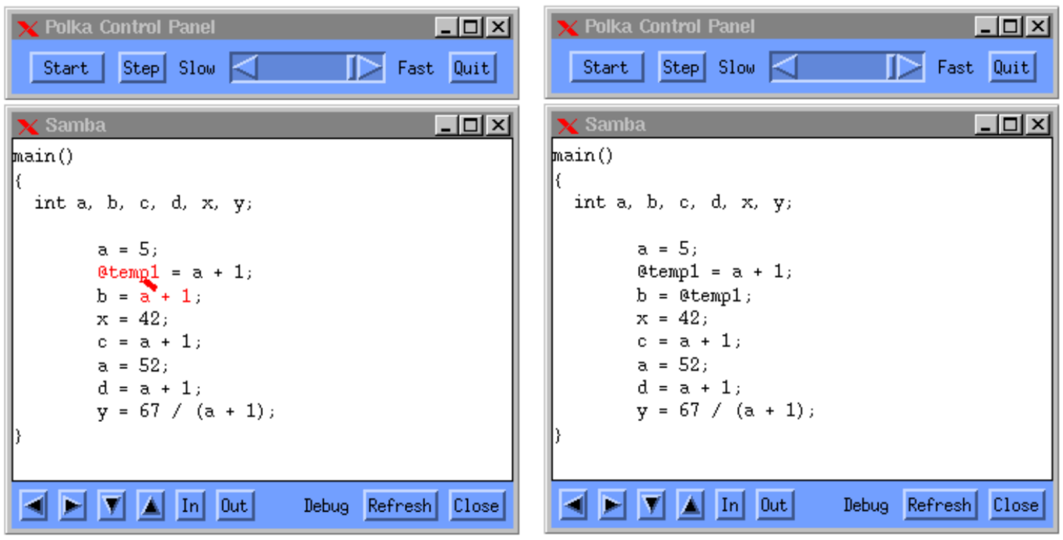
3.2. CLaX

CLaX is a visualized compiler developed by Sander et al. (1995). It visualizes internal data structures after different compiler phases. These are, for example, a syntax tree, a control flow graph, and a data dependency graph. It supports different layouts to visualize graphs. For example, a Manhattan layout can be used where all edges consist of horizontal or vertical line segments. To overcome the problem of limited screen size, a fisheye view can be used that distorts large graphs to fit inside the viewport (Sander et al., 1995).

3.3. The Feedback Compiler

The feedback compiler was developed by Binkley et al. (1998) and provides information from a compiler's backend. This information is subsequently used for visualization. Figure 3.2 shows the feedback compiler during common subexpression elimination.

Binkley et al. (1998) conducted a controlled experiment to measure whether visualization helps students understand common subexpression elimination. First, all students had to attend a 10-minute long video lecture on common subexpression elimination, followed by pre-test to measure their understanding. Afterwards, all students had to read a prepared text about common subexpression elimination for 25 minutes. They were then divided into an experimental group and a control group. The control group continued reading and worked on some code examples, while the experimental group used the feedback compiler to examine some code examples for 30 minutes.



```
main()
{
  int a, b, c, d, x, y;

        a = 5;
        @temp1 = a + 1;
        b = a + 1;
        x = 42;
        c = a + 1;
        a = 52;
        d = a + 1;
        y = 67 / (a + 1);
}
```

(a) Before elimination

```
main()
{
  int a, b, c, d, x, y;

        a = 5;
        @temp1 = a + 1;
        b = @temp1;
        x = 42;
        c = a + 1;
        a = 52;
        d = a + 1;
        y = 67 / (a + 1);
}
```

(b) After elimination

Figure 3.2.: Screenshot of the feedback compiler during a common subexpression elimination (Binkley et al., 1998)

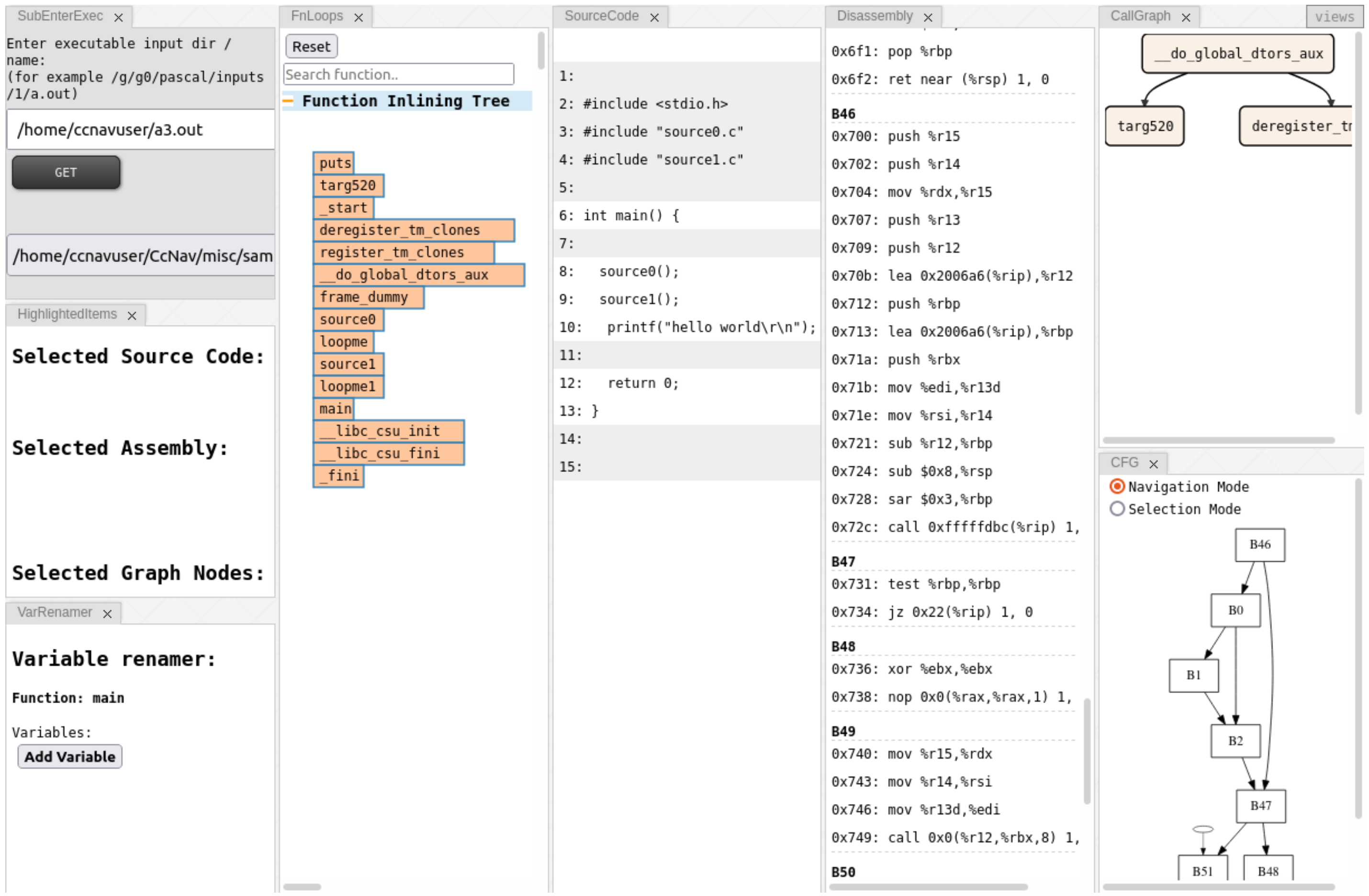
Finally, all students completed a post-test in which three questions were carried over from the pre-test, three questions were about already known code examples, and three questions were completely new to all students.

Binkley et al. (1998) concluded that there was no significant difference between the two groups and that students using the feedback compiler performed just as good as those who read only the written material. Another finding was that the experimental group had more confidence in their newly acquired knowledge than the control group.

3.4. CcNav

Compilation Navigator (CcNav) is a browser-based client-server application developed by Devkota et al. (2020). CcNav does not optimize code by itself. Instead, it tries to visualize what an external optimizer has done. Therefore, it needs a compiled binary file and its corresponding source code file as input. Then, the correlation between the original source code and the disassembled code of the executable file is displayed. For this purpose, multiple views are used, such as an original source code view, a control flow graph view, and a disassembled code view, as shown in Figure 3.3 on the following page. Users can then navigate through CcNav on their own. When a user clicks on an element in one view, the other views also switch to the corresponding program location.

CcNav was developed together with experts in the field of high-performance computing. To run it, we need to follow the installation instructions using Docker on the project page (Devkota et al., 2019).



SubEnterExec

```
Enter executable input dir /
name:
(for example /g/g0/pascal/inputs
/1/a.out)
```

/home/ccnavuser/a3.out

GET

/home/ccnavuser/CcNav/misc/sam

HighlightedItems

Selected Source Code:

Selected Assembly:

Selected Graph Nodes:

VarRenamer

Variable renamer:

Function: main

Variables:

Add Variable

FnLoops

Reset

Search function..

Function Inlining Tree

puts
targ520
_start
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
source0
loopme
source1
loopme1
main
__libc_csu_init
__libc_csu_fini
_fini

SourceCode

```c

#include <stdio.h>
#include "source0.c"
#include "source1.c"

int main() {

   source0();
   source1();
   printf("hello world\r\n");

   return 0;
}


```

Disassembly

```
0x6f1: pop %rbp
0x6f2: ret near (%rsp) 1, 0
B46
0x700: push %r15
0x702: push %r14
0x704: mov %rdx,%r15
0x707: push %r13
0x709: push %r12
0x70b: lea 0x2006a6(%rip),%r12
0x712: push %rbp
0x713: lea 0x2006a6(%rip),%rbp
0x71a: push %rbx
0x71b: mov %edi,%r13d
0x71e: mov %rsi,%r14
0x721: sub %r12,%rbp
0x724: sub $0x8,%rsp
0x728: sar $0x3,%rbp
0x72c: call 0xfffffdbc(%rip) 1,
B47
0x731: test %rbp,%rbp
0x734: jz 0x22(%rip) 1, 0
B48
0x736: xor %ebx,%ebx
0x738: nop 0x0(%rax,%rax,1) 1,
B49
0x740: mov %r15,%rdx
0x743: mov %r14,%rsi
0x746: mov %r13d,%edi
0x749: call 0x0(%r12,%rbx,8) 1,
B50
```

CallGraph

views

CFG

[✓ Navigation Mode] Selection Mode

Figure 3.3.: Screenshot of CcNav (Devkota et al., 2020)

4. Implementation

From the user's perspective, the application has to perform the following tasks:

1. Let the user input arbitrary programs (see Section 4.1 on the next page)
2. Visualize the program (see Section 4.2 on page 26)
3. Let the user specify optimization parameters (see Section 4.3 on page 28)
4. Visualize the optimization (see Section 4.4 on page 28)

According to the requirements defined in Section 1.1 on page 2, the resulting application has to be a website used by students to learn about code optimization. Therefore, we need a web server to serve static Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript files. This web server is written in Kotlin (JetBrains s.r.o., 2011) and uses Spring (VMware Inc., 2002) to handle Hypertext Transfer Protocol (HTTP) requests.

To reduce the workload on the server, we implemented the business logic for code optimization and visualization on the client side. Nowadays, there exist a variety of JavaScript libraries for building applications on the client side. React (Meta Platforms Inc., 2013) is one of them and was chosen because it already exists for many years and has a large community behind it.

Since JavaScript is not a strongly typed programming language, the entire client logic is implemented in TypeScript (Microsoft Corporation, 2012) and Kotlin. Nevertheless, the client source code needs to be compiled to JavaScript because browsers do not understand these languages. Advantageously, many type errors are already found during this compilation.

4.1. Getting the User's Program

Users who visit our website automatically land on the input page shown in Figure 4.1 on the facing page. The input page consists of a large text area where users can enter arbitrary Jova code and a sidebar on the right containing a general description of Jova.

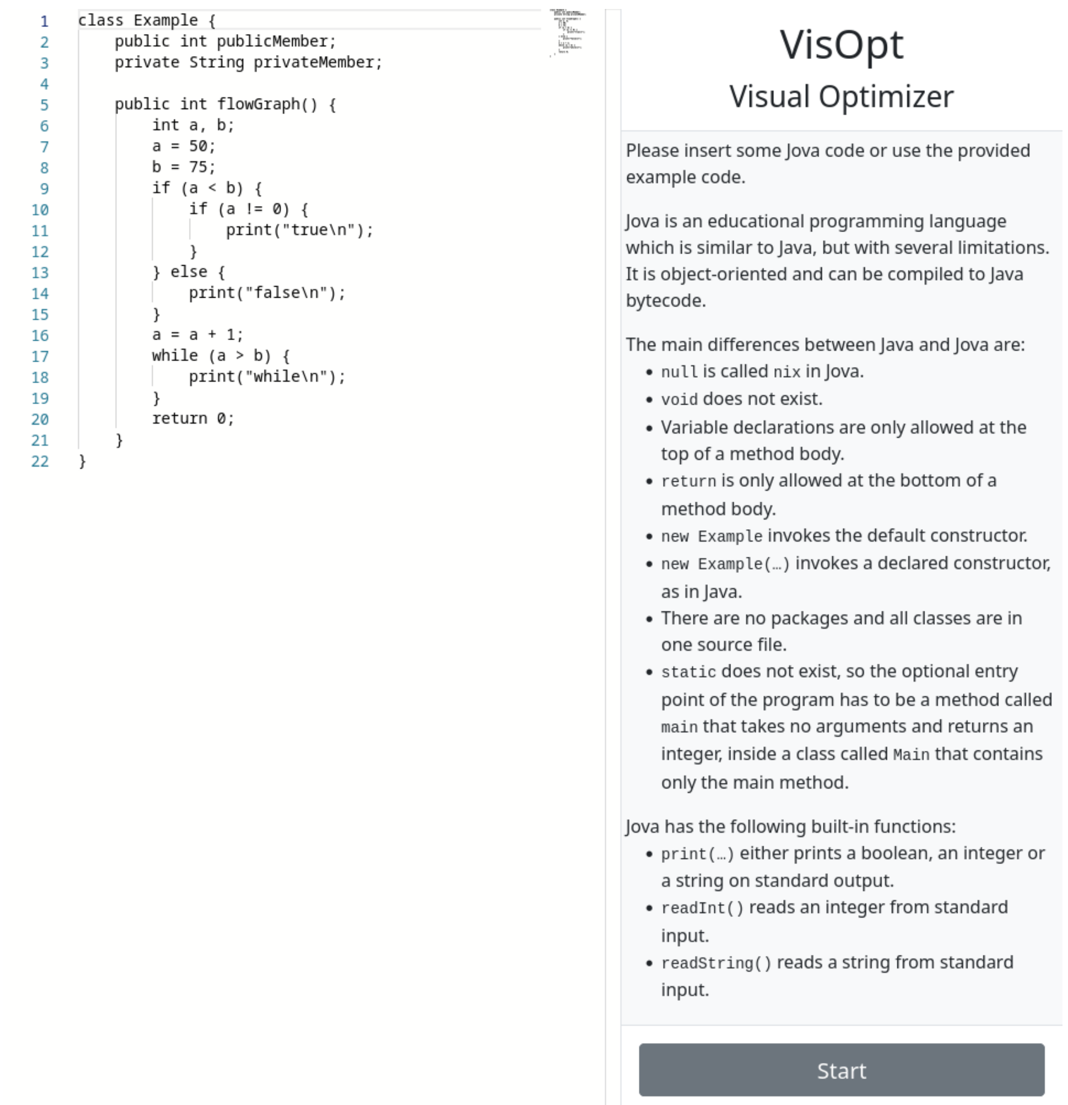
Jova is an object-oriented programming language similar to Java. It was designed for educational purposes and has some differences to Java. There are no packages and all classes and their methods are in a single source file. Every method has to return a value by having a return statement at the end of a method body. The declaration of local variables is only allowed at the beginning of a method body.

Since VisOpt has a built-in compiler, users only need to provide the source code as input, unlike CcNav (Devkota et al., 2020), where users also need to provide the optimized machine code.

As suggested by Saraiya et al. (2004), the text area already contains a sample program to help new users get started quickly. Users familiar with Visual Studio Code will recognize the look and feel of the text area as both use the Monaco Editor (Microsoft Corporation, 2015).

The sidebar also contains a start button. When we click the start button, the content of the text area is sent to the server. This in turn performs the lexical, syntactical, and semantical analysis of the program using ANTLR (Parr, 1992). In case of an error, the server responds an error report containing an error message and the exact code location that caused the error. Otherwise, the server responds with the syntax tree of the program.

While the client is waiting for the server's response, a loading indicator is displayed. In case of an error response, the error report is shown on the sidebar and the user has to fix the errors before they can continue. Otherwise, the input page gets replaced by the visualization page where the program is visualized.



```
class Example {
    public int publicMember;
    private String privateMember;

    public int flowGraph() {
        int a, b;
        a = 50;
        b = 75;
        if (a < b) {
            if (a != 0) {
                print("true\n");
            }
        } else {
            print("false\n");
        }
        a = a + 1;
        while (a > b) {
            print("while\n");
        }
        return 0;
    }
}
```

VisOpt

Visual Optimizer

Please insert some Jova code or use the provided example code.

Jova is an educational programming language which is similar to Java, but with several limitations. It is object-oriented and can be compiled to Java bytecode.

The main differences between Java and Jova are:

- `null` is called `nix` in Jova.
- `void` does not exist.
- Variable declarations are only allowed at the top of a method body.
- `return` is only allowed at the bottom of a method body.
- `new Example` invokes the default constructor.
- `new Example(…)` invokes a declared constructor, as in Java.
- There are no packages and all classes are in one source file.
- `static` does not exist, so the optional entry point of the program has to be a method called `main` that takes no arguments and returns an integer, inside a class called `Main` that contains only the main method.

Jova has the following built-in functions:

- `print(…)` either prints a boolean, an integer or a string on standard output.
- `readInt()` reads an integer from standard input.
- `readString()` reads a string from standard input.

Figure 4.1.: Screenshot of the input page of VisOpt

4.2. Visualize the Program

The visualization page consists of a visualization area displaying a selected method and a sidebar on the right, as shown in Figure 4.2b on the next page. Using the sidebar, the user can return to the input page, redo or undo optimizations, change the view of the program, and start an optimization. The application can view the selected method as flow graph or as code.

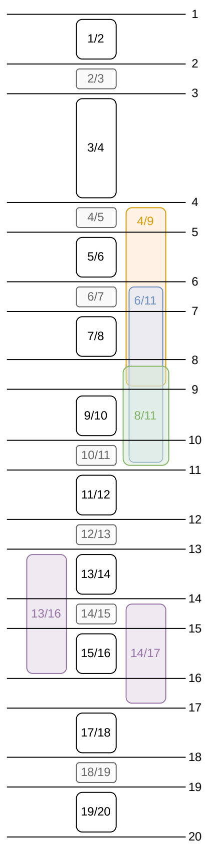
4.2.1. Flow Graph View

The flow graph view is the default view and uses a node-link representation to show the control flow graph of the selected method.

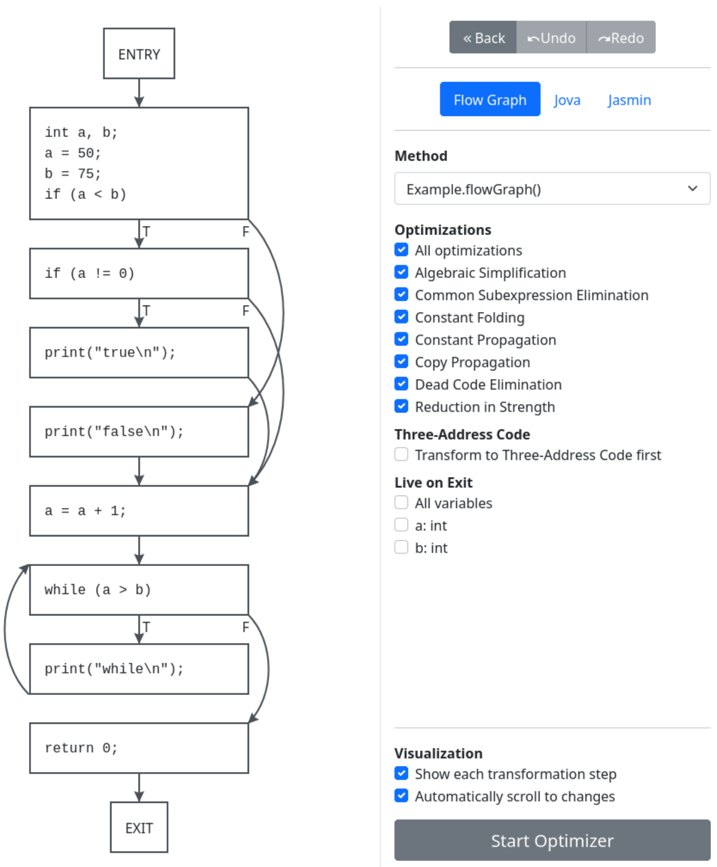
Nowadays there are many third-party libraries to draw graphs on websites. In contrast to these, one could calculate the positions of nodes and arrows by hand. However, we choose a middle way between these two extremes.

There exists a relatively new CSS layout called *grid* (Atkins et al., 2020) that is supported by most modern browsers. We use a $3 \times 2n$ grid, where n is the number of nodes in our flow graph. In the first column we place our upward pointing arrows. The second column contains our nodes and straight arrows. In the third column we place our downward pointing arrows. To draw the arrows, we use Scalable Vector Graphics (SVG). By intelligent merging and stacking of cells over each other, we can draw the flow graph while the browser does most of the position calculations for us. The only calculation we have to do ourselves is the curvature of the arrows. In Figure 4.2 on the facing page we see an example of the underlying grid layout and the resulting flow graph. Just like in XVPODB (Boyd and Whalley, 1993), all nodes have the same width and contain code instead of short labels as in CcNav (Devkota et al., 2020).

This approach makes us independent of third-party libraries and at the same time relieves us of the need to perform pixel-precise calculations for different screen sizes.



(a) Grid layout



```
int a, b;
a = 50;
b = 75;
if (a < b)
```

```
if (a != 0)
```

```
print("true\n");
```

```
print("false\n");
```

```
a = a + 1;
```

```
while (a > b)
```

```
print("while\n");
```

```
return 0;
```

Method

Example.flowGraph()

Optimizations

- [x] All optimizations
- [x] Algebraic Simplification
- [x] Common Subexpression Elimination
- [x] Constant Folding
- [x] Constant Propagation
- [x] Copy Propagation
- [x] Dead Code Elimination
- [x] Reduction in Strength

Three-Address Code

- [ ] Transform to Three-Address Code first

Live on Exit

- [ ] All variables
- [ ] a: int
- [ ] b: int

Visualization

- [x] Show each transformation step
- [x] Automatically scroll to changes

(b) Screenshot

Figure 4.2.: Grid layout and screenshot of the visualization page using the flow graph view

4.2.2. Code Views

There are two code views supported by the application: Jova and Jasmin. In the Jova view, the selected method is displayed as Jova code. Using this view, users can easily recognize their input code just like the source code view in CcNav, because it looks almost the same as on the input page.

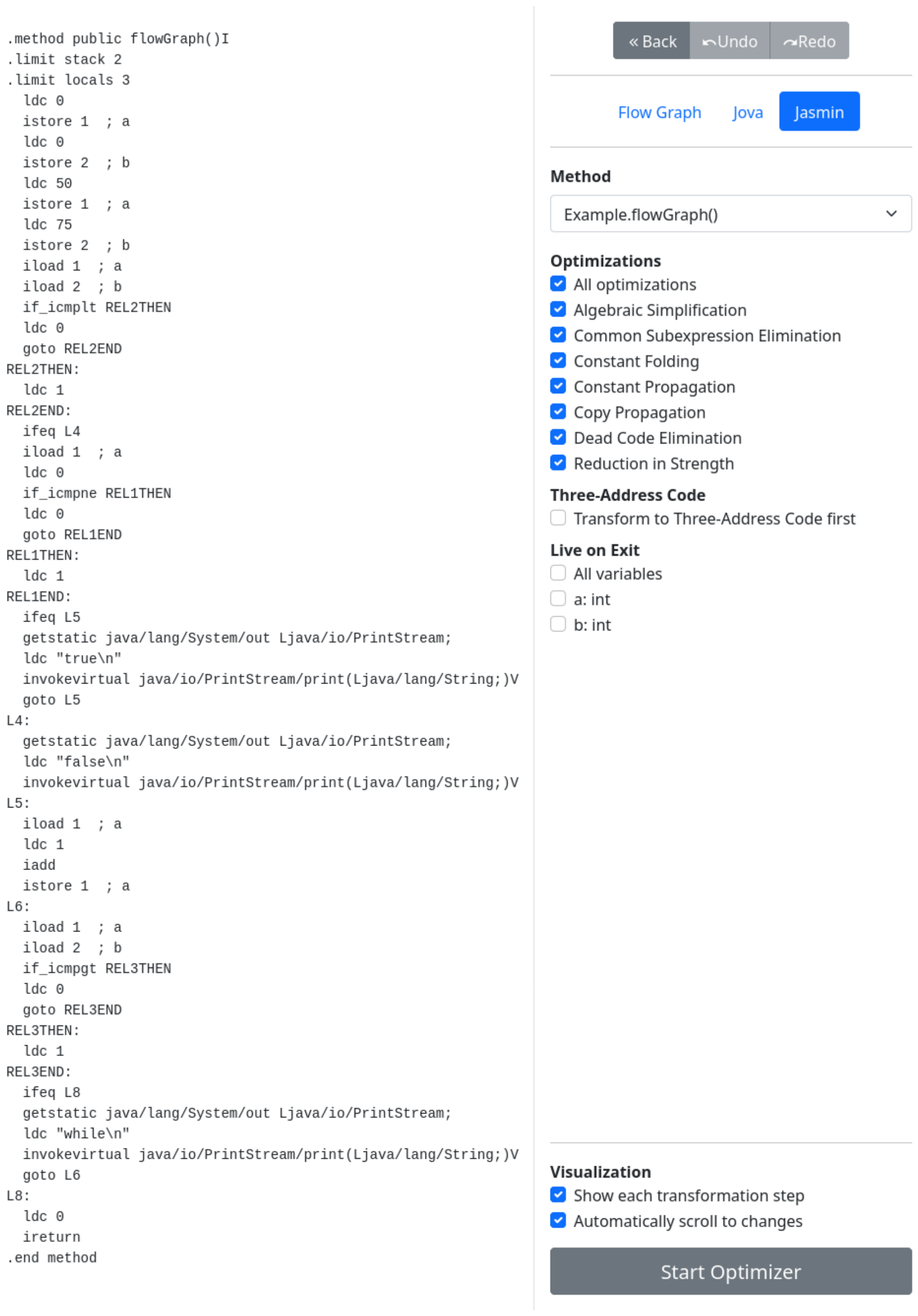
The Jasmin view, on the other hand, shows the compiled assembly code like the disassembly view in CcNav. Jasmin (Meyer, 1996) is an assembly language for Java bytecode that uses a local array for local variables and an operand stack instead of registers. The view is shown in Figure 4.3 on the next page.

4.3. Getting the User's Optimization Parameters

The optimizer needs to know the name of the method the user wants to optimize, which optimizations are enabled, and which variables are live on exit. Since three-address code is a commonly used intermediate representation in compilers, there is also an option to visualize the transformation to three-address code. Setting certain variables live on exit affects the live-variable analysis described on page 12, because IN[EXIT] is initialized with these variables. Users can set these parameters in the sidebar of the visualization page. Clicking on the start button activates the interactive visualization of the optimization.

4.4. Visualize the Optimization

This is the most challenging task that our application has to perform. Depending on the input program and optimization parameters, we want to show how the program is optimized. We have to visualize seven different optimization techniques (Algebraic Simplification, Common Subexpression Elimination, Constant Folding, Constant Propagation, Copy Propagation, Dead Code Elimination, and Reduction in Strength) which can be enabled



```
.method public flowGraph()I
.limit stack 2
.limit locals 3
  ldc 0
  istore 1  ; a
  ldc 0
  istore 2  ; b
  ldc 50
  istore 1  ; a
  ldc 75
  istore 2  ; b
  iload 1  ; a
  iload 2  ; b
  if_icmplt REL2THEN
  ldc 0
  goto REL2END
REL2THEN:
  ldc 1
REL2END:
  ifeq L4
  iload 1  ; a
  ldc 0
  if_icmpne REL1THEN
  ldc 0
  goto REL1END
REL1THEN:
  ldc 1
REL1END:
  ifeq L5
  getstatic java/lang/System/out Ljava/io/PrintStream;
  ldc "true\n"
  invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V
  goto L5
L4:
  getstatic java/lang/System/out Ljava/io/PrintStream;
  ldc "false\n"
  invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V
L5:
  iload 1  ; a
  ldc 1
  iadd
  istore 1  ; a
L6:
  iload 1  ; a
  iload 2  ; b
  if_icmpgt REL3THEN
  ldc 0
  goto REL3END
REL3THEN:
  ldc 1
REL3END:
  ifeq L8
  getstatic java/lang/System/out Ljava/io/PrintStream;
  ldc "while\n"
  invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V
  goto L6
L8:
  ldc 0
  ireturn
.end method
```

Method

Example.flowGraph()

Optimizations

- [✓] All optimizations
- [✓] Algebraic Simplification
- [✓] Common Subexpression Elimination
- [✓] Constant Folding
- [✓] Constant Propagation
- [✓] Copy Propagation
- [✓] Dead Code Elimination
- [✓] Reduction in Strength

Three-Address Code

- [ ] Transform to Three-Address Code first

Live on Exit

- [ ] All variables
- [ ] a: int
- [ ] b: int

Visualization

- [✓] Show each transformation step
- [✓] Automatically scroll to changes

Figure 4.3.: Screenshot of the visualization page using the Jasmin view

or disabled individually. Some of those use internal data structures that also need to be visualized.

4.4.1. Compiler Construction

Before we can start with the visualization we have to implement a compiler as already described in Section 2.1 on page 5. The source language of our compiler is Jova. Just implementing all compiler phases is not sufficient because we also have to visualize what happens during the optimization phase. We can achieve this by recording information while it performs the optimization and use this information later for visualization (Binkley et al., 1998).

As already mentioned, we want to keep the workload on the server side as low as possible. Therefore, we will perform the analysis part and intermediate code generation phase on the server side. The code optimization and generation phase take place in the browser on the client side. To send the intermediate representation to the client, we need to add two additional compiler phases: a serialization and deserialization phase.

Our compiler translates into multiple target languages, since there are three different views. Therefore, having a well-designed intermediate representation is essential.

4.4.2. Intermediate Representation

Our intermediate representation must be able to be visualized as a control flow graph and translated into Jasmin and Jova code. To translate it into Jova code, our intermediate representation has to preserve as much information as possible from the source code. Therefore, it is hierarchically designed like a syntax tree:

- A *program* consists of classes.
- *Classes* consist of constructors, members, and methods.
- Each *constructor* or *method* has a body.
- A *body* has exactly one compound.

- *Compounds* consist of compound statements.
- Each *compound statement* can be either an if-then-else statement, while statement, return statement, or a basic block.
- An *if-then-else statement* consists of a basic block and one or two compounds representing the branches.
- A *while statement* consists of a basic block and one compound, while a *return statement* only consists of a basic block.
- Each *basic block* consists of basic statements.
- *Basic statements* can be either an assignment or a bare expression.
- An *expression* may consist of sub expressions.

We can easily program such a hierarchy using Kotlin's sealed classes. With Kotlin's serialization library (JetBrains s.r.o., 2017), we can simply encode and decode instances of our intermediate representation as JavaScript Object Notation (JSON) and send them over a network connection.

4.4.3. Visualization

To visualize a sequence of optimization steps, the compiler has to record commands while it performs the optimization (Binkley et al., 1998). Analyzing all optimization techniques led to the conclusion that we can visualize them all with six commands:

- Add basic statement: Adds a statement at a specified position in a basic block.
- Remove basic statement: Removes a statement at a specified position in a basic block.
- Remove compound statement: Removes a statement at a specified position in a compound.
- Replace basic statement: Replaces a statement with another statement at a specified position in a basic block.
- Replace expression: Replaces an expression with another expression within a basic statement or expression.
- Take branch: Replaces an if-then-else statement with a branch.

Each optimization is a sequence of these commands. In Table 4.1 on the next page we can see which optimization requires which commands.

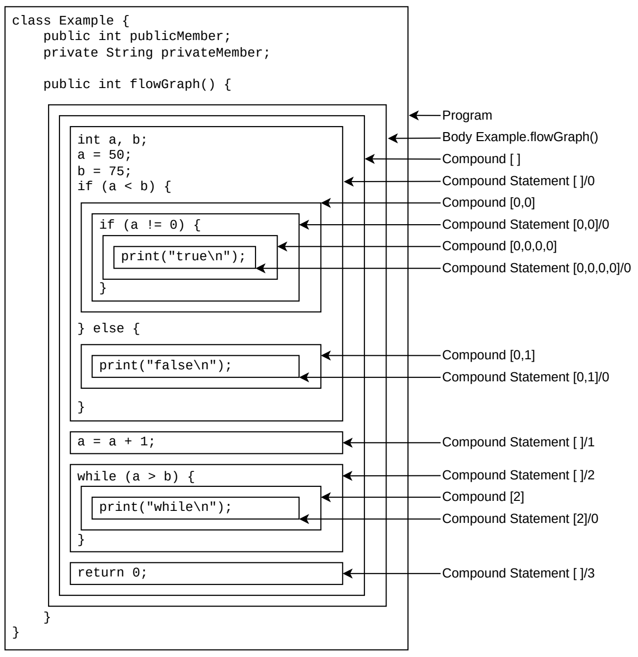
Command \ Transformation	Algebraic Simplification	Common Subexpression Elimination	Constant Folding	Constant Propagation	Copy Propagation	Dead Code Elimination	Reduction in Strength	Transform to Three-Address Code
Add basic statement		✓						✓
Remove basic statement						✓		
Remove compound statement						✓		
Replace basic statement						✓		
Replace expression	✓	✓	✓	✓	✓		✓	✓
Take branch						✓		

Table 4.1.: Command usage of code transformations

Each command requires parameters that specify what was changed in the intermediate representation and the location of the change. We need to implement a command executor to avoid having to keep a copy of the intermediate representation in memory after each command. To remember where a change occurred in our intermediate representation, each entity needs to be addressable. An address consists of the type of the entity and a unique combination of indices leading to the entity. An example of how entities are addressed is show in Figure 4.4 on the facing page.

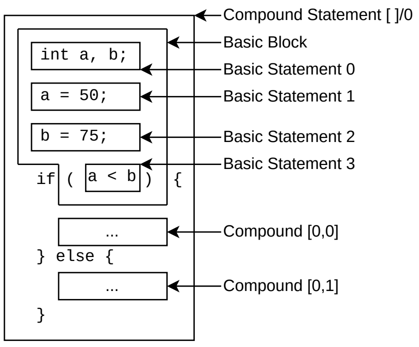
The address information is also used to highlight a code change. To make it easier to track changes, the visualization first scrolls to the position where a change is going to take place. Only in the next step the change is applied. This divides each command into a before and after step, turning our sequence of commands into a sequence of steps.

To help the user understand why a particular code change was made, we display the exact reason and current state of an important internal data structure used by the optimizer on the right-hand side of the screen.



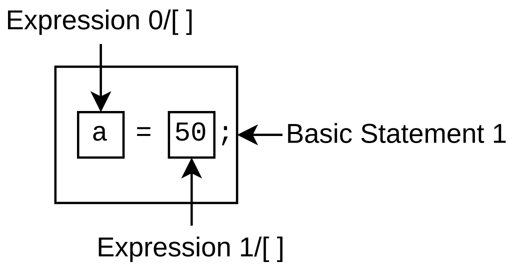
```
class Example {
    public int publicMember;
    private String privateMember;

    public int flowGraph() {
        int a, b;
        a = 50;
        b = 75;
        if (a < b) {
            if (a != 0) {
                print("true\n");
            }
        } else {
            print("false\n");
        }
        a = a + 1;
        while (a > b) {
            print("while\n");
        }
        return 0;
    }
}
```

(a) Program

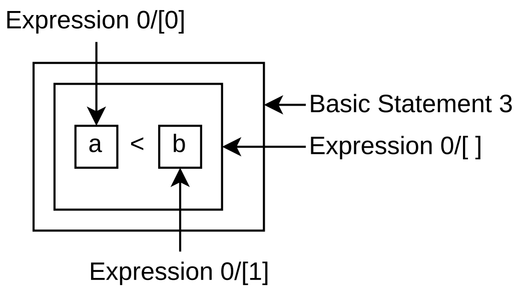


```
int a, b;
a = 50;
b = 75;
if ( a < b ) {
    ...
} else {
    ...
}
```

(b) Compound Statement [ ]/0



(c) Basic Statement 1



(d) Basic Statement 3

Figure 4.4.: Example addresses of entities in the intermediate representation

Figure 4.5 on the next page shows how a command caused by a constant propagation is visualized. The symbol table, which maps variables to values, is an important data structure during constant propagation and is shown in the sidebar on the right.

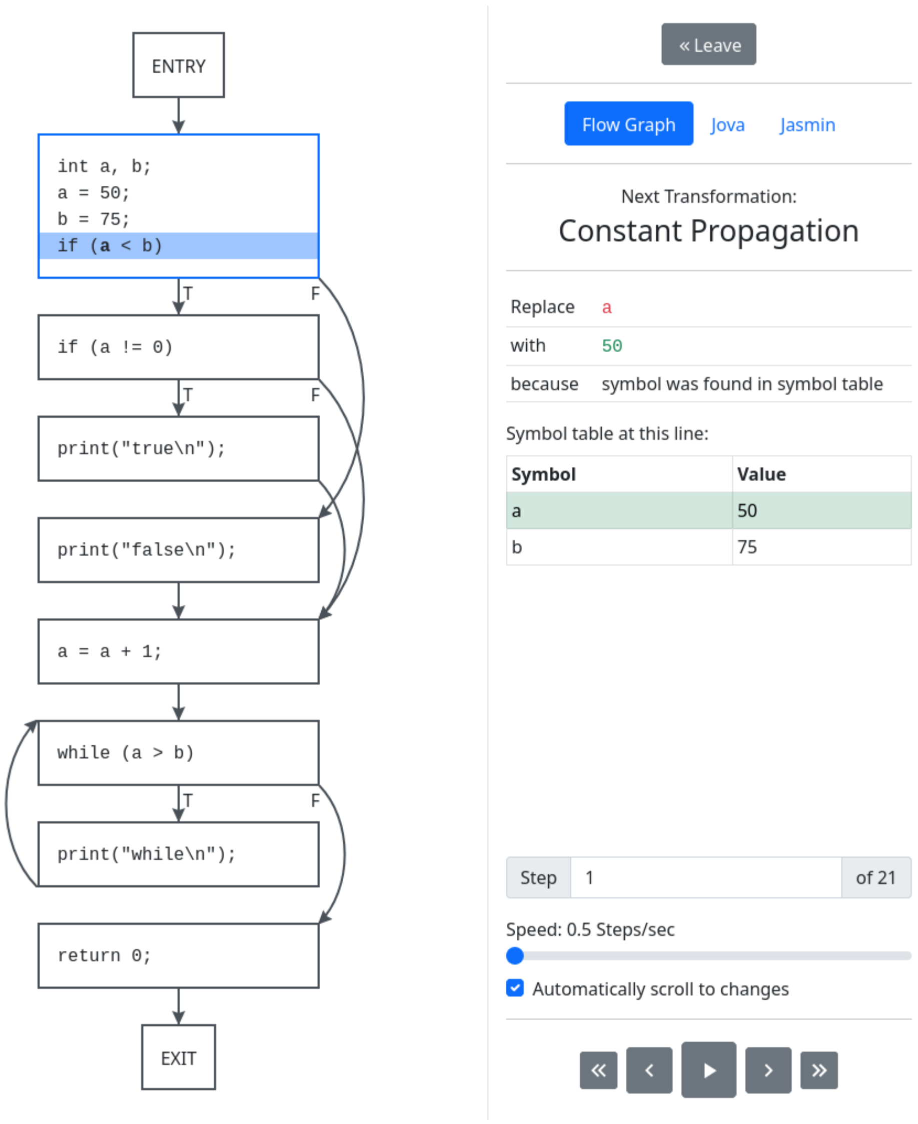
4.4.4. User Control

Users should be able to navigate through a sequence of steps at their own pace, as they learn at different rates (Saraiya et al., 2004). Therefore, the user can navigate through the sequence of steps using pagination buttons. Clicking on the step forward (›) or step backward (‹) button, the application jumps to the previous or next step, respectively. It is also possible to jump to the first or last step by clicking the continue forward (») or continue backward («) button. These buttons allow users to repeat the sequence until they fully understand the optimization performed by the compiler (Binkley et al., 1998).

To meet the requirements, we also need to implement a mode where the application automatically jumps through the sequence of steps at a certain pace. Therefore, a play/pause button and a speed slider were added to control automatic jumping.

4.5. Testing

To test the compiler extensively, we prepared 2150 test cases from different sources. 1739 test cases were written by the study assistants of the course "Compiler Construction" at Graz University of Technology in the summer semester 2022. 318 test cases were written by the students of group 1 of the "Compiler Construction" practicals at Graz University of Technology in the summer semester 2020. The remaining 93 test cases were written by the author of this thesis. Using JUnit (The JUnit Team, 2000), the compiler gets automatically tested, leading to a code coverage of 87% of all lines.



```
int a, b;
a = 50;
b = 75;
if (a < b)
```

```
if (a != 0)
```

```
print("true\n");
```

```
print("false\n");
```

```
a = a + 1;
```

```
while (a > b)
```

```
print("while\n");
```

```
return 0;
```

Next Transformation:

Constant Propagation

Replace	a
with	50
because	symbol was found in symbol table

Symbol table at this line:

Symbol	Value
a	50
b	75

Step 1 of 21

Speed: 0.5 Steps/sec

☑ Automatically scroll to changes

Figure 4.5.: Screenshot of the visualization page during an optimization using the flow graph view

4.6. Structure of the Project

The source code of VisOpt is publicly available at

`https://github.com/franzmandl/visopt`.

The repository contains the source code of the compiler and the web client. Since most of the test cases, especially those written by study assistants, were not allowed to be published, there exists also a private repository containing these test cases.

The compiler is a Kotlin Multiplatform (JetBrains s.r.o., 2022) project that contains three software modules. A server module contains the lexical, syntactical, and semantical analysis and runs on the Java Virtual Machine. Another module compiles to JavaScript and serves as an interface between Kotlin and TypeScript code. The largest module contains code shared by client and server, such as the intermediate representation, the code optimizer, and the code generator. It is also responsible for recording visualization commands during code optimization.

The web client contains the user interface and is based on Create-React-App (Meta Platforms Inc., 2016). It depends on some modules of the compiler project, because it needs the recorded visualization commands for visualization. An overview of all software modules is shown in Figure 4.6 on the facing page.

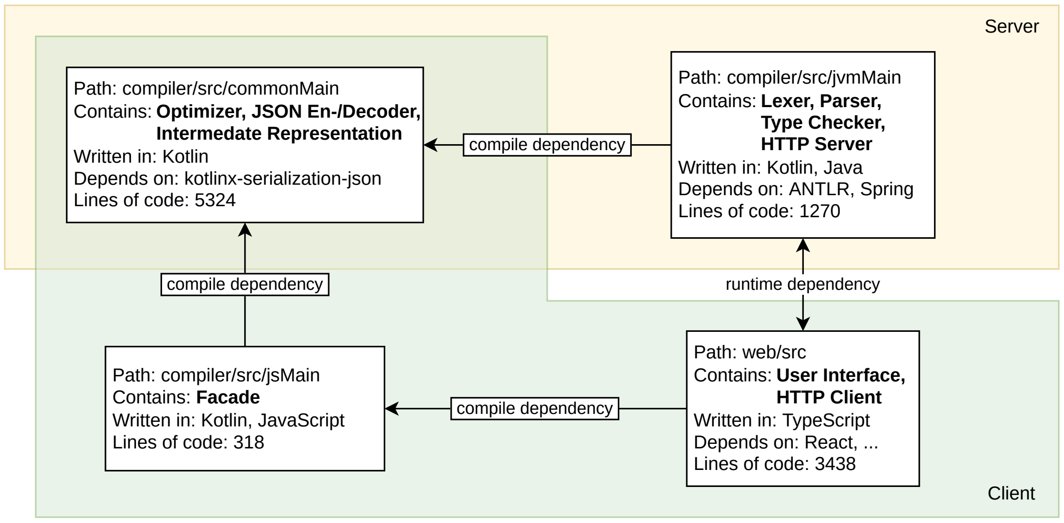

Server
Path: compiler/src/commonMain
Contains: Optimizer, JSON En-/Decoder, Intermediate Representation
Written in: Kotlin
Depends on: kotlinx-serialization-json
Lines of code: 5324
compile dependency
Path: compiler/src/jvmMain
Contains: Lexer, Parser, Type Checker, HTTP Server
Written in: Kotlin, Java
Depends on: ANTLR, Spring
Lines of code: 1270
compile dependency
runtime dependency
Path: compiler/src/jsMain
Contains: Facade
Written in: Kotlin, JavaScript
Lines of code: 318
compile dependency
Path: web/src
Contains: User Interface, HTTP Client
Written in: TypeScript
Depends on: React, ...
Lines of code: 3438
Client

Figure 4.6.: Dependencies between the software modules of VisOpt

5. Experiment

This chapter describes the design and evaluation of an experiment conducted for this thesis. The goal of the experiment was to determine whether using VisOpt significantly improves students' understanding of code optimization.

5.1. Design

For our experiment, we use a pre-test-post-test design as described by Dimitrov and Jr. (2003) and performed by Binkley et al. (1998). Generally, a pre-test-post-test design is used to measure the change between two points in time resulting from a treatment. Therefore, a sample of individuals is divided into several groups, and all groups perform a pre-test and a post-test. Between pre- and post-test, at least one group is exposed to the treatment and compared to at least one control group that did not receive the treatment. The difference between the groups of the pre- and post-test results can then be analyzed using statistical methods (Dimitrov and Jr., 2003).

5.2. Participants

The participant in our experiment were students attending the Compiler Construction course at Graz University of Technology. Participating in the experiment was voluntary, but participants received bonus points for their upcoming Compiler Construction exam. The entire experiment was conducted online. By completing the online pre-test, students were automatically registered for the experiment. Only after the pre-test, the participants were randomly divided into two groups. The experimental group used VisOpt, while the control group did not. Since only three students did not participate in the corresponding Compiler Construction practicals at Graz University of Technology, they were not considered for the final result. We also filtered students who aborted a test.

Finally, we obtained 53 valid participants, of which 26 were in the experimental group and 27 in the control group.

5.3. Timetable

The experiment was conducted between April 25, 2022 and May 3, 2022 as shown in Table 5.1 . Each activity takes about 10 to 30 minutes. Participants were allowed to do the activities whenever and for as long as they wanted within the specified time period. The timetable was strict and there were no exceptions for late submissions.

From	*To*	*Activity*
April 25, 2022	April 26, 2022	All participants performed the pre-test.
April 27, 2022		All participants were randomly divided into two groups.
April 28, 2022	April 29, 2022	All groups performed their training activity.
May 2, 2022	May 3, 2022	All participants performed the post-test.

Table 5.1.: Timetable of the experiment

5.4. Pre- and Post-Test

The goal of the pre-test is to get a baseline measure of the students' understanding of the topic (Binkley et al., 1998). The pre-test is attached in Appendix A on page 51 and it consists of nine questions. Two questions are surveys asking participants to assess their own level of competence in code optimization and feedback on the pre-test. The remaining seven questions are graded and relate to code optimization techniques.

The post-test is attached in Appendix E on page 71 and contains the same question about the level of competence as well as the seven graded questions about code optimization techniques. Additionally, we also ask participants to provide feedback on the training activity and how much they think their understanding improved.

The seven graded questions consist of four multiple-choice questions: one about constant folding, one about common subexpression elimination, and two about dead code elimination. The remaining three questions are open-ended questions that require participants to solve a coherent optimization example using constant propagation, constant folding, and dead code elimination, in this order.

As for grading, all graded questions are worth the same. When scoring multiple-choice questions, we assume a baseline score of 50% and make a weighted deduction for each incorrect choice and a weighted addition for each correct choice, resulting in a possible score between 0% and 100% for each question.

We assume a baseline score of 100% when scoring open-ended questions and make a deduction for each mistake, with 0% being the minimum. We have also considered that some open-ended questions depend on the previous answer. 124 pre- and post-tests were submitted, resulting in 45 different ways to answer the three open-ended questions.

The final score on a test is a percent value between 0% to 100%.

5.5. Training Activities

For the training activities, we have prepared a summary on the theory of code optimization, which is attached in Appendix B on page 57. It is an early draft of Section 2.2 on page 6. Each group got a different training activity.

The experimental group was instructed to read through the code optimization theory and visualize the code examples after each section using VisOpt. The instruction sheet is attached in Appendix C on page 63.

The control group's training activity consisted of reading through the code optimization theory and understand the code examples without VisOpt. The instruction sheet is attached in Appendix D on page 69.

5.6. Results

In this section, we analyze whether we need to accept or reject our null hypothesis "*There is no statistically significant difference between the two groups*" based on the pre- and post-test scores.

First, we calculate the gain scores by subtracting the pre-test score from the post-test score of each participant (Dimitrov and Jr., 2003). In Table 5.2 on the facing page we can see the average score of the experimental group on each question as well as the average gain score. In Table 5.3 on the next page we can see the same for the control group. We can see that the experimental group performed slightly better in the pre- and post-test than the control group.

In the next step, we check our null hypothesis using Analysis of Variance (ANOVA) (Ståhle and Wold, 1989). ANOVA can be used to analyze two or more groups for statistical significance by testing whether the variance between the groups is greater than the variance within the groups. If there is statistical significance, then the grouping was meaningful and there is a difference between the groups. In our case, we have only one random variable to analyze, the gain scores, and therefore use a single-factor (also called one-way) ANOVA. ANOVA gives us a P-value between 0 and 1. The P-value is the probability of obtaining test results that are at least as extreme as the actual observed result, assuming that the null hypothesis is correct (Wasserstein and Lazar, 2016). For statistical significance, we usually need a P-value below 0.05 (Ståhle and Wold, 1989).

Question	*Pre-Test*	*Gain*	*Post-Test*
Constant Folding	75.32%	+3.53%	78.85%
Common Subexpression Elimination	77.88%	0.00%	77.88%
Dead Code Elimination (1/2)	73.08%	+6.73%	79.81%
Dead Code Elimination (2/2)	91.35%	+6.73%	98.08%
Open-ended (three questions)	69.89%	+14.94%	84.83%
All graded questions	75.33%	+8.83%	84.16%

Table 5.2.: Achieved scores of the experimental group

Question	*Pre-Test*	*Gain*	*Post-Test*
Constant Folding	67.90%	+0.62%	68.52%
Common Subexpression Elimination	73.15%	+4.63%	77.78%
Dead Code Elimination (1/2)	64.81%	+10.19%	75.00%
Dead Code Elimination (2/2)	92.59%	-1.85%	90.74%
Open-ended (three questions)	69.90%	+5.82%	75.72%
All graded questions	72.59%	+4.44%	77.03%

Table 5.3.: Achieved scores of the control group

Our analysis of the gain scores between pre- and post- test yielded to a P-value of 0.2991. We also analyzed each question individually, as shown in Table 5.4 on the following page, but there is not statistical significance.

Therefore, we have to accept the null hypothesis and conclude that the understanding about code optimization of students who read about code optimization theory and visualize the examples with VisOpt improved the same as of students who only read about it.

5.7. Survey

The pre- and post-test allowed to provide feedback and an estimation on the code optimization competence level. In this section, we discuss this data.

Adding up all the options for written feedback, it was used in 38% of the cases. Many participants stated in the pre-test that they had not yet studied

Question	*P-value*	*Calculation*
Constant Folding	0.6088	see Table F.3 on page 80
Common Subexpression Elimination	0.3503	see Table F.4 on page 80
Dead Code Elimination (1/2)	0.7440	see Table F.5 on page 80
Dead Code Elimination (2/2)	0.1264	see Table F.6 on page 81
Open-ended (three questions)	0.2414	see Table F.7 on page 81
All graded questions	0.2991	see Table F.8 on page 81

Table 5.4.: ANOVA results of the experiment

code optimization. Some saw the pre-test as good practice for the practicals and upcoming exam.

In both groups, participants seemed to enjoy the training activities. The experimental group had fun playing around with VisOpt while reading through the theory, as indicated by some participants:

> "I really liked the interactive examples and the compact description in the document."

> "The visual optimizer is great. It makes more fun, playing around with that, then reading the theory. It also helps to understand the theory."

The Jasmin view was mentioned twice as useful for the practicals:

> "I like this type of visualization very much, especially that you can also look at the flow graph, the Jova code and (most importantly) the Jasmin assembly."

> "I would have loved to have this website during the practicals to understand the implementation of optimizations. Like that the Jasmin code is also available."

One participant has already saved VisOpt as a bookmark for exam preparation. There was also a feature request for using the arrow keys on the keyboard to navigate through the optimization steps, which was implemented afterwards.

Six participants in the control group found the summary about code optimization easy to understand with clear and concise examples. On the other hand, there were two demands for more complex examples in the summary. Two participants indicated on the post-test that the theory did not stick in their memory very well:

> “I hardy remember anything from the PDF. With the PDF I would solve all the task, but without its kind of impossible.”

> “When I read through it 2 days ago it all made sense, but without any materials right now it was hard for me to remember everything exactly.”

One participant criticized that the post-test was quite short and that the questions may not reflect student knowledge accurately enough. This could be one of the reasons why we have to accept the null hypothesis. It is also possible that some of the questions were too difficult or that the grading of the questions led to this result. Another reason could be that the training activities were too simple and too short, as some participants indicated: “Simple but good activity”

In Figure 5.1 on the next page we see the result for estimating the level of competence. The estimated level of competence was almost the same in both groups at the pre-test. In the post-test, the experimental group seems to be slightly more confident about their level of competence. No one in either group reported having a high level of competence, neither on the pre-test nor on the post-test. Except for one participant in the control group, all participants reported the same or a higher level of competence on the post-test than on the pre-test, although the gain score of this particular participant increased by 27%. In some cases, the level of competence estimation is clearly not accurate, as the gain score has decreased in some cases.

In the experimental group, all participants either agreed or strongly agreed that the training activity improved their understanding of code optimization, as shown in Figure 5.2 on the following page, while only 70% of the control group agreed or strongly agreed with this statement.

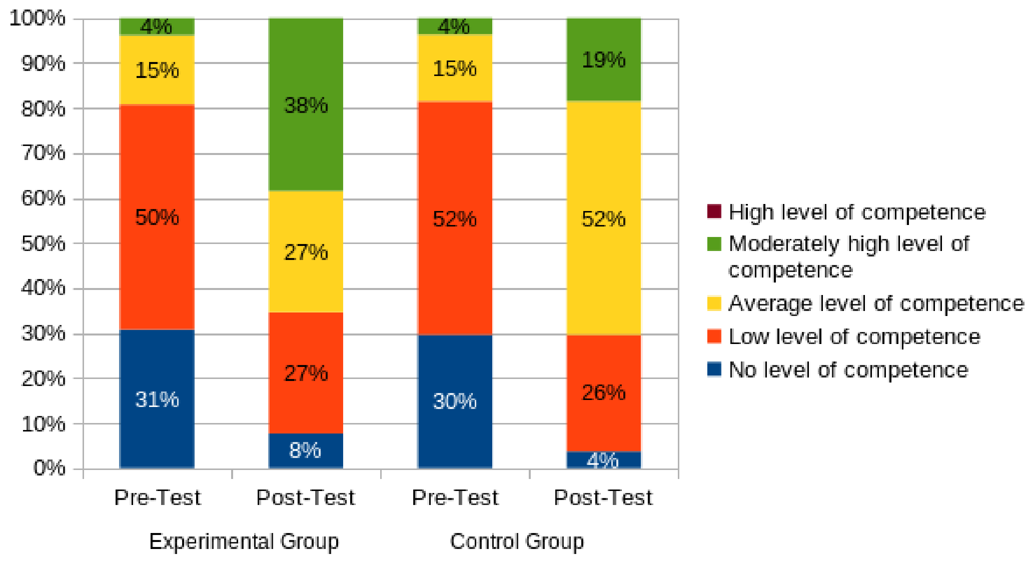


Figure 5.1.: Participants' estimation of their level of competence

All survey results as well as the gain scores for each participant are listed in the appendix. For the experimental group, see Table F.1 on page 78 and for the control group, see Table F.2 on page 79.

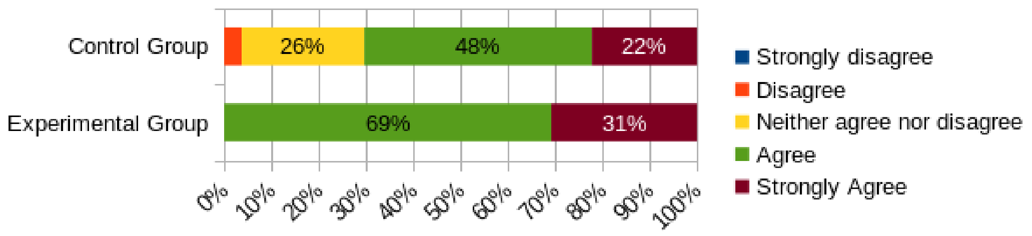


Figure 5.2.: Participants' agreement that the training activity improved their understanding of code optimization

6. Conclusion

In this thesis, we introduced a new web application for visualizing compiler optimizations called VisOpt. The application takes code written in a Java-like language called Jova as input and uses various representation methods for visualization, such as a flow graph or a Java bytecode assembly language. At the time of writing, the web application was accessible at

`http://ccvisual.ist.tugraz.at/`.

It does not require any software to be installed by the user other than a modern web browser. Navigation buttons allow users to undo and redo specific optimization steps. The source code of the project is publicly available at

`https://github.com/franzmandl/visopt`.

The project includes a powerful yet extensible compiler with an optimizer written in Kotlin, and web client using state-of-the-art web technologies such as React and TypeScript.

6.1. Experiment Results

An experiment was conducted using a pre-test-post-test design to determine if using the application to learn significantly improves the understanding of code optimization. No such improvement was detected, except those students who used the application were slightly more confident in their level of competence. The same results were also obtained by Binkley et al. (1998) with a similar application called *The Feedback Compiler*.

6.2. Future Work

Machine-Dependent Optimization. In the scope of this thesis we only focused on machine-independent optimization, while machine-dependent optimization can also be visualized. A machine-dependent peephole optimization to eliminate redundant Jasmin stack instructions has already been implemented and tested, but not visualized.

More Optimization Techniques. VisOpt currently fully supports seven optimization techniques and the transformation to three-address code. Implementing other optimization techniques would be a good way of extension, since many other techniques exist. Some of them are also described by Aho et al. (2006), and Tremblay and Sorenson (1985).

More Source Languages. The educational language *Jova* is at the moment the only source language accepted by VisOpt. An extension possibility would be to improve the grammar of Jova to be more similar to Java, or to add a whole new source language like Python.

Visualization Improvements. There are several ways to improve visualization. One possibility would be to make the views more interactive. For example, if the user clicks on the edge of the flow graph, the visualization jumps to the corresponding basic block as in XVPODB (Boyd and Whalley, 1993). A bigger improvement would be to make the Jova code view editable and merge it with the input page.

Appendix

Appendix A.

Pre-Test

Please perform the test to the best of your ability without using any additional materials.

Question 1: Background

Did you participate in the Compiler Construction practicals this semester or in any previous semester?

- ◯ Yes
- ◯ No

How familiar are you with code optimization? Please estimate your current skills on a scale from 1 (= No level of competence) to 5 (= High level of competence).

- ◯ 1 (= No level of competence)
- ◯ 2 (= Low level of competence)
- ◯ 3 (= Average level of competence)
- ◯ 4 (= Moderately high level of competence)
- ◯ 5 (= High level of competence)

Question 2: Constant Folding

Check all examples for **Constant Folding** replacements. « and » are bit shift operations.

- ☐ a. replacing `301 + 402` with `703`
- ☐ b. replacing `4 * 3` with `3 « 2`
- ☐ c. replacing `5 * 2` with `10`
- ☐ d. replacing `8 » 1` with `4`
- ☐ e. replacing `a * 1` with `a`

Question 3: Common Subexpression Elimination

Consider the following basic block. Check all expressions which are good candidates for **Common Subexpression Elimination**.

```
a = 50;
b = 4 * a;
c = a / 2;
a = a / 2;
d = 4 * a;
```

- ☐ a. `50`
- ☐ b. `4 * a`
- ☐ c. `a / 2`

Question 4: Dead Code Elimination (1/2)

Consider the following basic block. By only applying **Dead Code Elimination**, which line(s) would be eliminated? There are no variables live on exit.

```
1. a = 3;
2. b = 7;
3. a = a + 1;
4. if (b < 9)
```

☐ a. line 1
☐ b. line 2
☐ c. line 3
☐ d. line 4

Question 5: Dead Code Elimination (2/2)

Consider the following basic block. By only applying **Dead Code Elimination**, which line(s) would be eliminated? Variables `f` and `g` are live on exit.

```
1. f = 3;
2. g = 6;
3. h = 9;
4. g = 2 * g;
5. f = h;
```

☐ a. line 1
☐ b. line 2
☐ c. line 3
☐ d. line 4
☐ e. line 5

Question 6: General (Step 1: Constant Propagation)

Consider the following basic block.

```
int a,b,c;
a = 21;
b = a / 3;
c = b + a;
return a * 2;
```

Optimize the basic block above using **Constant Propagation**!

TODO: Change this with your solution
int a,b,c;
a = 21;
b = a / 3;
c = b + a;
return a * 2;

Question 7: General (Step 2: Constant Folding)

Optimize the result of step 1 using **Constant Folding**!

Question 8: General (Step 3: Dead Code Elimination)

Optimize the result of step 2 using **Dead Code Elimination** and consider variable `b` live on exit!

Question 9

Do you have any comments/remarks on the test?

Appendix B.

Code Optimization Theory

The goal of code optimization is to generate improved code that performs better on certain objectives. For example, some objectives might be code size, execution speed, or power consumption (Aho et al., 2006). The behavior of the program must be preserved during code optimization.

In code optimization, we differentiate between local and global optimization.

B.1. Local and Global Optimization

Local optimization improves code within a maximum sequence of consecutive instructions where the flow of control can only

(a) enter the sequence through the first instruction of the sequence and
(b) leave only through the last instruction of the sequence.

Such a sequence of instructions is called a basic block (Aho et al., 2006). Only the last instruction in a basic block is allowed to be a halting, branching, or jumping instruction.

While local optimization only takes into account what happens inside a basic block, global optimization on the other hand considers what happens across basic blocks, e.g., by using a control flow graph.

B.2. Control Flow Graph

A control flow graph is a directed graph where nodes represent basic blocks and edges represent possible transfers of control between these basic blocks (Shivers, 1991). To indicate the flow of control's start and end we can add two special nodes labeled ENTRY and EXIT.

The corresponding control flow graph of Listing B.1 is illustrated in Figure B.1 using a node-link representation.

```
int a, b;
a = 50;
b = 75;
if (a < b) {
    if (a != 0) {
        print("true");
    }
} else {
    print("false");
}
a = a + 1;
while (a > b) {
    print("while");
}
return 0;
```

Listing B.1: Example code written in Jova

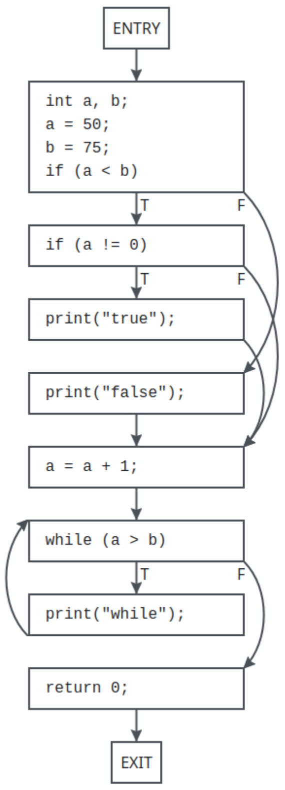


Figure B.1.: Node-link representation of the control flow graph of Listing B.1

B.3. Algebraic Simplification

Algebraic simplification is an optimization technique that uses algebraic laws to simplify code. For example, we use algebraic identities, such as

```
x + 0 = x          0 + x = x
x - 0 = x
x * 1 = x          1 * x = x
x / 1 = x
```

to improve our code (Aho et al., 2006).

B.4. Common Subexpression Elimination

By finding common subexpressions we can eliminate unnecessary computations evaluating to the same values (Aho et al., 2006). By introducing temporary variables, we store the results of those computations. For example, applying this technique on Listing B.2 results in Listing B.3 .

```
a = 3 * 4 / 2;
b = 3 * 4 * 2;
result = (a + 10) * (a + 10);
```

Listing B.2: Example code before common subexpression elimination

```
tmp1 = 3 * 4;
a = tmp1 / 2;
b = tmp1 * 2;
tmp2 = a + 10;
result = tmp2 * tmp2;
```

Listing B.3: Example code after common subexpression elimination

B.5. Constant Folding

Folding constants means that constant expressions are evaluated at compile time and replaced with their constant value (Aho et al., 2006). For example, we may replace the expression $2 * 3 + 4$ with 10.

B.6. Constant Propagation

We can replace variable references by constant values if that variable has a unique constant value at that point in the code (Aho et al., 2006). For example, in Listing B.4 we may replace variable `i` with value 2 resulting in Listing B.5 .

```
i = 2;
result = i * i + i;
```

Listing B.4: Example code before constant propagation

```
i = 2;
result = 2 * 2 + 2;
```

Listing B.5: Example code after constant propagation

B.7. Copy Propagation

The idea behind copy propagation is that after an assignment (or copy) statement `u = v`, we may use `v` over `u` afterwards (Aho et al., 2006). For example, using variable `i` over `copy` in Listing B.6 results in Listing B.7 on the facing page.

```
copy = i;
result = copy * copy + copy;
```

Listing B.6: Example code before copy propagation

```
copy = i;
result = i * i + i;
```

Listing B.7: Example code after copy propagation

B.8. Dead Code Elimination

Eliminating instructions that compute values that are never used is called *dead code elimination* (Aho et al., 2006). A variable is *dead* at a point in a program, if the value of a variable is not accessed before the next time it gets assigned. We may eliminate assignments to dead variables. If the value of a variable may get accessed later in another basic block, we call it *live on exit*.

Using dead code elimination on Listing B.8 with variable `b` live on exit eliminates the first assignment of `b` as well as the assignment of `c` as shown in Listing B.9 .

```
a = 201;
b = 302;
c = a * 2;
b = a + 3;
```

Listing B.8: Example code before dead code elimination

```
a = 201;
b = a + 3;
```

Listing B.9: Example code after dead code elimination

Unreachable code is code that never gets executed under any circumstances at runtime (Debray et al., 2000). Therefore, we can eliminate it during compile time. For example, in Listing B.10 on the next page we may eliminate the else branch.

```
if (true) {
    print("always true");
} else {
    print("unreachable else branch");
}
```

Listing B.10: Example code before unreachable code elimination

B.9. Reduction in Strength

Replace more expensive operators by cheaper ones is called *reduction in strength* (Aho et al., 2006). For example, a multiplication or division by a power of two can be replaced with a cheaper addition or bit shift operation, such as:

```
x * 2 = x + x         2 * x = x + x
x * 4 = x « 2         4 * x = x « 2
x / 4 = x » 2
```

Appendix C.

Experimental Group's Training Activity

Please make a serious attempt when following the tasks of the training activity.

- Read the first two pages about code optimization in the attached file *Code Optimization* (Appendix B on page 57).
- Afterwards, visit the optimization visualizer VisOpt (`http://ccvisual.ist.tugraz.at/`) and click the Start-button at the bottom right.

Question 1

- Read Section B.3 about **Algebraic Simplification** in the Code Optimization file.
- Afterwards, select method **Example.algebraicSimplification()** in VisOpt and disable all optimizations except Algebraic Simplification.
- Click on the "Start Optimizer"-button at the bottom right. Use the navigation buttons at the bottom right to navigate between the transformation steps and try to improve your understanding about code optimization.
- Click the Leave-button at the top right when you are done.

Have you completed the task? ◯ Yes ◯ No

Question 2

- Read Section B.4 about **Common Subexpression Elimination** in the Code Optimization file.
- Afterwards, select method **Example.commonSubexpressionElimination()** in VisOpt and disable all optimizations except Common Subexpression Elimination.
- As before, click the "Start Optimizer"-button and navigate through the transformation steps while improving your understanding about code optimization.
- Click the Leave-button when you are done.

Have you completed the task? ◯ Yes ◯ No

Question 3

- Read Section B.5 about **Constant Folding** in the Code Optimization file.
- Afterwards, select method **Example.constantFolding()** in VisOpt and disable all optimizations except Constant Folding.
- As before, click the "Start Optimizer"-button and navigate through the transformation steps while improving your understanding about code optimization.
- Click the Leave-button when you are done.

Have you completed the task? ◯ Yes ◯ No

Question 4

- Read Section B.6 about **Constant Propagation** in the Code Optimization file.
- Afterwards, select method **Example.constantPropagation()** in VisOpt and disable all optimizations except Constant Propagation.
- As before, click the "Start Optimizer"-button and navigate through the transformation steps while improving your understanding about code optimization.

- Click the Leave-button when you are done.

Have you completed the task? ◯ Yes ◯ No

Question 5

- Read Section B.7 about **Copy Propagation** in the Code Optimization file.
- Afterwards, select method **Example.copyPropagation()** in VisOpt and disable all optimizations except Copy Propagation.
- As before, click the "Start Optimizer"-button and navigate through the transformation steps while improving your understanding about code optimization.
- Click the Leave-button when you are done.

Have you completed the task? ◯ Yes ◯ No

Question 6

- Read Section B.8 about **Dead Code Elimination** in the Code Optimization file.
- Afterwards, select method **Example.deadCodeElimination()** in VisOpt and disable all optimizations except Dead Code Elimination.
- **Set variable `b` live on exit.**
- As before, click the "Start Optimizer"-button and navigate through the transformation steps while improving your understanding about code optimization.
- Click the Leave-button when you are done.

Have you completed the task? ◯ Yes ◯ No

Question 7

- Select method **Example.unreachableCodeElimination()** in VisOpt and disable all optimizations except Dead Code Elimination.

- As before, click the "Start Optimizer"-button and navigate through the transformation steps while improving your understanding about code optimization.
- Click the Leave-button when you are done.

Have you completed the task? ◯ Yes ◯ No

Question 8

- Read Section B.9 about **Reduction in Strength** in the Code Optimization file.
- Afterwards, select method **Example.reductionInStrength()** in VisOpt and disable all optimizations except Reduction in Strength.
- As before, click the "Start Optimizer"-button and navigate through the transformation steps while improving your understanding about code optimization.
- Click the Leave-button when you are done.

Have you completed the task? ◯ Yes ◯ No

Question 9

- Have a look at method **Example.allOptimizations()** and optimize it by enabling all optimizations.
- Finally, you can play around with VisOpt as you like.
- If you want, you can also optimize your own code by going back to the code input.

Have you completed the task? ◯ Yes ◯ No

Question 10

Do you think you have understood Code Optimization?

- ◯ Yes
- ◯ Not Completely
- ◯ No

Question 11

Do you have any additional comments on the training activity?

Appendix D.

Control Group's Training Activity

Please make a serious attempt when following the tasks of the training activity.

Question 1

Read the theory about code optimization in the attached file *Code Optimization* (Appendix B on page 57) and try to follow the examples.

Have you read the file?

◯ Yes
◯ No

Do you think you have understood the theory?

◯ Yes
◯ Not Completely
◯ No

Question 2

Do you have any additional comments on the training activity?

Appendix E.

Post-Test

Please perform the test to the best of your ability without using any additional materials.

Question 1: Constant Folding

Check all examples for **Constant Folding** replacements. « and » are bit shift operations.

- ☐ a. replacing `301 + 402` with `703`
- ☐ b. replacing `4 * 3` with `3 « 2`
- ☐ c. replacing `5 * 2` with `10`
- ☐ d. replacing `8 » 1` with `4`
- ☐ e. replacing `a * 1` with `a`

Question 2: Common Subexpression Elimination

Consider the following basic block. Check all expressions which are good candidates for **Common Subexpression Elimination**.

```
a = 50;
b = 4 * a;
c = a / 2;
a = a / 2;
d = 4 * a;
```

☐ a. `50`
☐ b. `4 * a`
☐ c. `a / 2`

Question 3: Dead Code Elimination (1/2)

Consider the following basic block. By only applying **Dead Code Elimination**, which line(s) would be eliminated? There are no variables live on exit.

1. `a = 3;`
2. `b = 7;`
3. `a = a + 1;`
4. `if (b < 9)`

☐ a. line 1
☐ b. line 2
☐ c. line 3
☐ d. line 4

Question 4: Dead Code Elimination (2/2)

Consider the following basic block. By only applying **Dead Code Elimination**, which line(s) would be eliminated? Variables `f` and `g` are live on exit.

1. `f = 3;`
2. `g = 6;`
3. `h = 9;`
4. `g = 2 * g;`
5. `f = h;`

- ☐ a. line 1
- ☐ b. line 2
- ☐ c. line 3
- ☐ d. line 4
- ☐ e. line 5

Question 5: General (Step 1: Constant Propagation)

Consider the following basic block.

```
int a,b,c;
a = 21;
b = a / 3;
c = b + a;
return a * 2;
```

Optimize the basic block above using **Constant Propagation**!

TODO: Change this with your solution
int a,b,c;
a = 21;
b = a / 3;
c = b + a;
return a * 2;

Question 6: General (Step 2: Constant Folding)

Optimize the result of step 1 using **Constant Folding**!

Question 7: General (Step 3: Dead Code Elimination)

Optimize the result of step 2 using **Dead Code Elimination** and consider variable `b` live on exit!

Question 8: Background

How familiar are you with code optimization? Please estimate your current skills on a scale from 1 (= No level of competence) to 5 (= High level of competence).

◯ 1 (= No level of competence)
◯ 2 (= Low level of competence)
◯ 3 (= Average level of competence)
◯ 4 (= Moderately high level of competence)
◯ 5 (= High level of competence)

Do you agree that the training activity improved your understanding of code optimization?

- ◯ 1. Strongly disagree
- ◯ 2. Disagree
- ◯ 3. Neither agree nor disagree
- ◯ 4. Agree
- ◯ 5. Strongly Agree

Question 9

Do you have any additional remarks about the training activity?

Appendix F.

Experiment Results

ID[1]	*Pre*[2]	*Post*[3]	*Agree*[4]	*CF*[5]	*CSE*[6]	*DCE1*[7]	*DCE2*[8]	*General*[9]	*All*[10]
1	1	4	5	0%	0%	0%	0%	+83%	+36%
2	3	4	5	0%	0%	+25%	+75%	+67%	+43%
3	2	4	5	0%	0%	0%	0%	-11%	-5%
4	2	4	5	0%	0%	-50%	+13%	0%	-5%
5	2	2	5	+25%	0%	0%	0%	+25%	+14%
6	2	2	4	-33%	0%	+75%	0%	-83%	-30%
7	2	3	4	0%	-25%	0%	0%	0%	-4%
8	4	4	5	0%	-25%	0%	0%	0%	-4%
9	1	3	4	0%	0%	0%	0%	-17%	-7%
10	2	3	4	+25%	0%	+25%	+63%	0%	+16%
11	1	1	4	+8%	0%	-25%	0%	+17%	+5%
12	2	2	4	-25%	0%	+75%	+63%	+22%	+26%
13	1	1	4	0%	+25%	+25%	0%	+11%	+12%
14	1	3	4	-17%	+50%	-50%	0%	+60%	+23%
15	2	2	4	+25%	0%	+25%	0%	0%	+7%
16	2	3	4	+25%	0%	+50%	-13%	+17%	+16%
17	2	2	4	-8%	0%	0%	0%	0%	-1%
18	2	2	4	0%	0%	0%	0%	+17%	+7%
19	3	4	4	-25%	0%	0%	0%	-17%	-11%
20	2	4	4	+33%	0%	+50%	+13%	+100%	+57%
21	3	4	5	+25%	0%	-50%	0%	+25%	+7%
22	1	3	4	+42%	0%	+25%	-13%	+37%	+23%
23	2	4	5	-17%	-25%	0%	0%	+8%	-2%
24	1	2	4	+8%	0%	0%	-25%	-6%	-5%
25	3	4	4	-25%	0%	0%	0%	0%	-4%
26	1	3	4	+25%	0%	-25%	0%	+33%	+14%

Table F.1.: Pre- and post-test results of the experimental group

[1]*ID* ... participant identifier

[2]*Pre* ... competence level estimation on pre-test (1 = No level of competence, 2 = Low level of competence, 3 = Average level of competence, 4 = Moderately high level of competence, 5 = High level of competence)

[3]*Post* ... competence level estimation on post-test (1 to 5 mean the same as for *Pre*[2])

[4]*Agree* ... agreement that the training activity has improved understanding (1 = Strongly disagree, 2 = Disagree, 3 = Neither agree nor disagree, 4 = Agree, 5 = Strongly Agree)

ID[1]	*Pre*[2]	*Post*[3]	*Agree*[4]	*CF*[5]	*CSE*[6]	*DCE1*[7]	*DCE2*[8]	*General*[9]	*All*[10]
27	2	4	3	+8%	0%	+75%	0%	-8%	+8%
28	2	3	4	+17%	+25%	+25%	0%	0%	+10%
29	2	2	5	0%	0%	0%	+13%	0%	+2%
30	3	4	4	0%	+25%	+25%	+13%	0%	+9%
31	1	3	5	+25%	+75%	+100%	0%	0%	+29%
32	1	2	3	+58%	-25%	-25%	-13%	+67%	+28%
33	2	3	4	-17%	0%	+25%	0%	+17%	+8%
34	4	4	4	0%	0%	-50%	0%	0%	-7%
35	2	2	3	0%	0%	0%	0%	+22%	+10%
36	2	3	4	+25%	0%	+25%	0%	-17%	0%
37	2	3	3	-25%	0%	+50%	0%	-17%	-4%
38	3	4	4	+25%	0%	0%	0%	+17%	+11%
39	2	3	5	0%	0%	0%	0%	0%	0%
40	2	3	4	-42%	0%	-50%	0%	0%	-13%
41	2	3	3	+17%	0%	+100%	0%	-17%	+10%
42	1	1	4	-25%	+50%	0%	0%	+27%	+15%
43	2	3	4	0%	0%	-50%	0%	0%	-7%
44	3	2	4	+17%	0%	+75%	0%	+33%	+27%
45	3	3	4	-42%	0%	0%	+38%	+17%	+7%
46	1	2	3	0%	+25%	-25%	0%	0%	0%
47	1	4	5	-17%	-25%	0%	0%	0%	-6%
48	1	2	3	0%	0%	-25%	-63%	0%	-12%
49	2	3	4	0%	0%	+50%	-38%	+17%	+9%
50	1	2	5	0%	0%	0%	0%	0%	0%
51	2	3	2	0%	0%	0%	0%	0%	0%
52	2	3	5	-8%	-25%	-50%	0%	0%	-12%
53	1	3	4	0%	0%	0%	0%	0%	0%

Table F.2.: Pre- and post-test results of the control group

[5]*CF* ... gain scores on the question “Constant Folding”
[6]*CSE* ... gain scores on the question “Common Subexpression Elimination”
[7]*DCE1* ... gain scores on the question “Dead Code Elimination (1/2)”
[8]*DCE2* ... gain scores on the question “Dead Code Elimination (2/2)”
[9]*General* ... gain scores on the open-ended questions
[10]*All* ... gain scores on all graded questions

Group	*Count*	*Sum*	*Mean*	*Variance*		
Experimental	26	0.9167	0.0353	0.0401		
Control	27	0.1667	0.0062	0.0443		
Source of Variation	*SS*[11]	*df*[12]	*MS*[13]	*F*[14]	*P-value*[15]	$F_{critical}$[16]
Between Groups	0.0112	1	0.0112	0.2652	0.6088	4.0304
Within Groups	2.1542	51	0.0422			
Total	2.1654	52				

Table F.3.: Single factor ANOVA on the question "Constant Folding"

Group	*Count*	*Sum*	*Mean*	*Variance*		
Experimental	26	0.0000	0.0000	0.0200		
Control	27	1.2500	0.0463	0.0434		
Source of Variation	*SS*[11]	*df*[12]	*MS*[13]	*F*[14]	*P-value*[15]	$F_{critical}$[16]
Between Groups	0.0284	1	0.0284	0.8885	0.3503	4.0304
Within Groups	1.6296	51	0.0320			
Total	1.6580	52				

Table F.4.: Single factor ANOVA on the question "Common Subexpression Elimination"

Group	*Count*	*Sum*	*Mean*	*Variance*		
Experimental	26	1.7500	0.0673	0.1078		
Control	27	2.7500	0.1019	0.1839		
Source of Variation	*SS*[11]	*df*[12]	*MS*[13]	*F*[14]	*P-value*[15]	$F_{critical}$[16]
Between Groups	0.0158	1	0.0158	0.1078	0.7440	4.0304
Within Groups	7.4771	51	0.1466			
Total	7.4929	52				

Table F.5.: Single factor ANOVA on the question "Dead Code Elimination (1/2)"

Group	*Count*	*Sum*	*Mean*	*Variance*		
Experimental	26	1.7500	0.0673	0.0540		
Control	27	-0.5000	-0.0185	0.0273		
Source of Variation	*SS*[11]	*df*[12]	*MS*[13]	*F*[14]	*P-value*[15]	$F_{critical}$[16]
Between Groups	0.0976	1	0.0976	2.4150	0.1264	4.0304
Within Groups	2.0605	51	0.0404			
Total	2.1580	52				

Table F.6.: Single factor ANOVA on the question "Dead Code Elimination (2/2)"

Group	*Count*	*Sum*	*Mean*	*Variance*		
Experimental	26	3.8833	0.1494	0.1283		
Control	27	1.5722	0.0582	0.0302		
Source of Variation	*SS*[11]	*df*[12]	*MS*[13]	*F*[14]	*P-value*[15]	$F_{critical}$[16]
Between Groups	0.1100	1	0.1100	1.4050	0.2414	4.0304
Within Groups	3.9928	51	0.0783			
Total	4.1028	52				

Table F.7.: Single factor ANOVA on the open-ended questions

Group	*Count*	*Sum*	*Mean*	*Variance*		
Experimental	26	2.2952	0.0883	0.0338		
Control	27	1.1976	0.0444	0.0131		
Source of Variation	*SS*[11]	*df*[12]	*MS*[13]	*F*[14]	*P-value*[15]	$F_{critical}$[16]
Between Groups	0.0256	1	0.0256	1.1007	0.2991	4.0304
Within Groups	1.1840	51	0.0232			
Total	1.2095	52				

Table F.8.: Single factor ANOVA on all graded questions

[11] *SS* ... sum of squares
[12] *df* ... degree of freedom
[13] *MS* ... mean square
[14] *F* ... test statistic
[15] *P-value* ... probability
[16] $F_{critical}$ ... critical value of F with a significance level of 0.05

Bibliography

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley (cit. on pp. 5–7, 9–14, 48, 57, 59–62).

Tab Atkins, Elika J. Etemad, Rossen Atanassov, and Oriol Brufau (2020). *CSS Grid Layout Module Level 1*. Available: `https://www.w3.org/TR/css-grid-1/` (visited on Jun 8, 2022) (cit. on p. 26).

Ronald M. Baecker and David Sherman (1981). *Sorting Out Sorting*. 16mm color sound film. Shown at SIGGRAPH '81, Dallas TX (cit. on p. 1).

Manuel E. Benitez and Jack W. Davidson (1988). "A portable global optimizer and linker." In: *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pp. 329–338 (cit. on p. 17).

David Binkley, Bruce Duncan, Brennan Jubb, and April Wielgosz (1998). "The feedback compiler." In: *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242)*. IEEE, pp. 198–205 (cit. on pp. 1, 20, 21, 30, 31, 34, 39, 40, 47).

Mickey Boyd and D. Whalley (1993). "Graphical visualization of compiler optimizations." MA thesis. Citeseer (cit. on pp. 1, 17–19, 26, 48).

Saumya Debray, William Evans, Robert Muth, and Bjorn De Sutter (2000). "Compiler Techniques for Code Compaction." In: *ACM Transactions on Programming Languages and Systems* 22, pp. 378–415 (cit. on pp. 13, 61).

Sabin Devkota, Pascal Aschwanden, Adam Kunen, Matthew Legendre, and Katherine E. Isaacs (2019). *LLNL/CcNav: CcNav is a web-based tool for visualizing compiler optimizations in binaries*. Available: `https://github.com/LLNL/CcNav` (visited on Jan. 24, 2022) (cit. on p. 21).

Sabin Devkota, Pascal Aschwanden, Adam Kunen, Matthew Legendre, and Katherine E. Isaacs (2020). "Ccnav: Understanding compiler optimizations in binary code." In: *IEEE transactions on visualization and computer graphics* 27.2, pp. 667–677 (cit. on pp. 2, 21, 22, 24, 26).

Dimiter M. Dimitrov and Phillip D. Rumrill Jr. (2003). “Pretest-posttest designs and measurement of change.” In: *Work* 20.2, pp. 159–165 (cit. on pp. 39, 42).

Florida State University (1993). *X-windows Very Portable Optimizer DeBugger*. Available: `ftp://ftp.cs.fsu.edu/pub/whalley/xvpodb` (visited on Jan. 21, 2022) (cit. on p. 19).

Eric Fouh, Monika Akbar, and Clifford A. Shaffer (2012). “The role of visualization in computer science education.” In: *Computers in the Schools* 29.1-2, pp. 95–117 (cit. on p. 1).

JetBrains s.r.o. (2011). *Kotlin Programming Language*. Available: `https://kotlinlang.org/` (visited on May 11, 2022) (cit. on p. 23).

JetBrains s.r.o. (2017). *Kotlin/kotlinx.serialization: Kotlin multiplatform / multi-format serialization*. Available: `https://github.com/Kotlin/kotlinx.serialization` (visited on Jun. 15, 2022) (cit. on p. 31).

JetBrains s.r.o. (2022). *Kotlin Multiplatform*. Available: `https://kotlinlang.org/docs/multiplatform.html` (visited on Jun. 17, 2022) (cit. on p. 36).

Meta Platforms Inc. (2013). *React – A JavaScript library for building user interfaces*. Available: `https://reactjs.org/` (visited on May 12, 2022) (cit. on pp. 2, 23).

Meta Platforms Inc. (2016). *facebook/create-react-app: Set up a modern web app by running one command.* Available: `https://github.com/facebook/create-react-app` (visited on Jun. 17, 2022) (cit. on p. 36).

Jonathan Meyer (1996). *Jasmin User Guide*. Available: `http://jasmin.sourceforge.net/guide.html` (visited on Jun 15, 2022) (cit. on p. 28).

Microsoft Corporation (2012). *TypeScript: JavaScript With Syntax For Types.* Available: `https://www.typescriptlang.org/` (visited on May 14, 2022) (cit. on p. 23).

Microsoft Corporation (2015). *Monaco Editor*. Available: `https://microsoft.github.io/monaco-editor/` (visited on Jun. 10, 2022) (cit. on p. 24).

Pálma Rozália Osztián, Zoltán Kátai, and Erika Osztián (2020). “Algorithm Visualization Environments: Degree of interactivity as an influence on student-learning.” In: *2020 IEEE Frontiers in Education Conference (FIE)*. IEEE, pp. 1–8 (cit. on p. 1).

Terence Parr (1992). *ANother Tool for Language Recognition*. Available: `https://www.antlr.org/` (visited on Jun 6, 2022) (cit. on pp. 2, 24).

Georg Sander, Martin Alt, Christian Ferdinand, and Reinhard Wilhelm (1995). "CLaX—A visualized compiler." In: *International Symposium on Graph Drawing*. Springer, pp. 459–462 (cit. on p. 19).

Purvi Saraiya, Clifford A. Shaffer, Scott McCrickard, and Chris North (2004). *Effective features of algorithm visualizations*. Vol. 36. 1. ACM (cit. on pp. 24, 34).

Olin Shivers (1991). "Control-flow analysis of higher-order languages." PhD thesis. Citeseer (cit. on pp. 7, 58).

Lars Ståhle and Svante Wold (1989). "Analysis of variance (ANOVA)." In: *Chemometrics and intelligent laboratory systems* 6.4, pp. 259–272 (cit. on p. 42).

John Stasko, Albert Badre, and Clayton Lewis (1993). "Do algorithm animations assist learning? An empirical study and analysis." In: *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pp. 61–66 (cit. on p. 1).

The JUnit Team (2000). *JUnit 5*. Available: `https://junit.org/` (visited on Jun. 16, 2022) (cit. on p. 34).

Paolo Tonella and Alessandra Potrich (2005). *Reverse Engineering of Object Oriented Code*. Springer Science & Business Media (cit. on p. 13).

Jean-Paul Tremblay and Paul G. Sorenson (1985). *Theory and Practice of Compiler Writing*. McGraw-Hill, Inc. (cit. on pp. 5, 48).

VMware Inc. (2002). *Spring Framework*. Available: `https://spring.io/` (visited on May 11, 2022) (cit. on pp. 2, 23).

Ronald L. Wasserstein and Nicole A. Lazar (2016). *The ASA statement on p-values: context, process, and purpose* (cit. on p. 42).

List of Figures

List of Tables

Acronyms

ANOVA Analysis of Variance. 40, 42, 78, 79
ANTLR ANother Tool for Language Recognition. 2, 24

CcNav Compilation Navigator. 21, 22, 24, 26, 28
CSS Cascading Style Sheets. 23, 26

DAG Directed Acyclic Graph. 9, 10

HTML Hypertext Markup Language. 23
HTTP Hypertext Transfer Protocol. 23

JSON JavaScript Object Notation. 31

RTL Register Transfer List. 17, 18

SVG Scalable Vector Graphics. 26

VisOpt Visual Optimizer. v, 2, 24, 25, 35–38, 40–42, 45, 46, 61–64
VPO Very Portable Optimizer. 17

XVPODB X-windows Very Portable Optimizer DeBugger. 17–19, 26, 46