



## Assignment of master's thesis

<b>Title:</b>	Ahead-of-time compiler for the microC language
<b>Student:</b>	Bc. Václav Král
<b>Supervisor:</b>	doc. Ing. Filip Kříkava, Ph.D.
<b>Study program:</b>	Informatics
<b>Branch / specialization:</b>	System Programming
<b>Department:</b>	Department of Theoretical Computer Science
<b>Validity:</b>	until the end of summer semester 2022/2023

### Instructions

The goal of this thesis is to develop an ahead-of-time compiler for the microC language that is used in the program analysis course (NI-APR). The aim is to have a compiler into which we can plug the results of the various static analysis that we cover in the first half of the course. The implementation should be easy to follow - clarity is preferred over conciseness so that it can be used as educational material. It should be written in Scala and compatible with the rest of the microC code base (interpreter and analysis). The code should be well tested and documented so the students can follow the code.





**FACULTY  
OF INFORMATION  
TECHNOLOGY  
CTU IN PRAGUE**

Master's thesis

# **Ahead-of-time compiler for the microC language**

*Bc. Václav Král*

Department of Theoretical Computer Science  
Supervisor: doc. Ing. Filip Křikava, Ph.D.

April 26, 2022



---

## **Acknowledgements**

First, I would like to thank my supervisor doc. Ing. Filip Křikava, Ph.D. for all his advice and time he dedicated to helping me with this thesis. I would also like to thank my friends and family who helped me a lot during my studies and provided the much needed psychological support.



---

## Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No.121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on April 26, 2022

.....

Czech Technical University in Prague  
Faculty of Information Technology  
© 2022 Václav Král. All rights reserved.

*This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).*

### **Citation of this thesis**

Král, Václav. *Ahead-of-time compiler for the microC language*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.



---

# Abstrakt

Cílem této práce je implementace ahead-of-time optimalizujícího kompilátoru *microC*—jazyku, který se používá v předmětu NI-APR (Vybrané metody analýzy programů) na FIT ČVUT pro výuku analýz programů. Tento kompilátor má především sloužit jako učební pomůcka předmětu NI-APR, která demonstruje uplatnění a užitečnost vybraných statických analýz při kompilaci a optimalizaci. V práci se čtenář seznámí nejen s architekturou kompilátorů, ale i s jednotlivými statickými analýzami, které kompilátor podporuje. Dále se práce zabývá návrhem a především implementací kompilátoru. Optimalizační schopnosti implementace jsou poté demonstrovány na několika ukázkových příkladech. Závěrem práce navrhuje možná zlepšení a rozšíření. Výsledkem práce je funkční optimalizující kompilátor jazyku *microC*.

**Klíčová slova** kompilátor, optimalizace, statická analýza kódu, *microC*, x86, Scala

---

# Abstract

The aim of this thesis is to implement an ahead-of-time optimizing compiler for *microC*—language used in the NI-APR (Selected Methods for Program Analysis) course at FIT CTU for teaching program analyses. The compiler should primarily serve as an educational material of the course NI-APR, which demonstrates application and usefulness of selected static analyses during compilation and optimization. In this thesis, the reader will get familiar with not only the architecture of compilers, but also with the static analyses supported by the compiler. Further in this thesis, the design and most importantly the implementation are discussed. The optimization capabilities of the implementation are then demonstrated on several examples. Some of the possible future work improvements are proposed at the end of the thesis. The result of the thesis is a working optimizing microC compiler.

**Keywords** compiler, optimization, static program analysis, microC, x86, Scala

---

# Contents

<b>Introduction</b>	<b>1</b>
<b>1 Background</b>	<b>3</b>
1.1 Compiler . . . . .	3
1.2 MicroC . . . . .	5
1.3 x86-64 . . . . .	9
1.4 Static program analysis . . . . .	11
<b>2 Design</b>	<b>17</b>
2.1 The frontend . . . . .	17
2.2 The middleend . . . . .	18
2.3 The backend . . . . .	26
<b>3 Implementation</b>	<b>41</b>
3.1 The frontend . . . . .	42
3.2 The middleend . . . . .	43
3.3 The backend . . . . .	48
3.4 Summary of the compiler . . . . .	55
3.5 Testing . . . . .	56
3.6 Documentation . . . . .	58
<b>4 Assessment</b>	<b>61</b>
4.1 Example 1 . . . . .	61
4.2 Example 2 . . . . .	64
4.3 Example 3 . . . . .	68
<b>Conclusion</b>	<b>71</b>
<b>Bibliography</b>	<b>73</b>

<b>A</b>	<b>Acronyms</b>	<b>75</b>
<b>B</b>	<b>Contents of enclosed SD card</b>	<b>77</b>
<b>C</b>	<b>MicroC</b>	<b>80</b>
<b>D</b>	<b>x86IR</b>	<b>82</b>
	D.1 Instruction set . . . . .	82
<b>E</b>	<b>Usage manual</b>	<b>87</b>
	E.1 How to use . . . . .	87
	E.2 How to run . . . . .	88
<b>F</b>	<b>Compiler output examples</b>	<b>91</b>

---

## List of Figures

1.1	A compiler. . . . .	3
1.2	The three stages of a compilation. . . . .	5
1.3	Typical memory layout of a running x86 program. . . . .	11
1.4	CFG of the program snippet from Listing 1.3. . . . .	13
2.1	The frontend of the microC compiler. . . . .	17
2.2	An optimizer. . . . .	20
2.3	An optimization process. . . . .	20
2.4	Example of dead code elimination. . . . .	23
2.5	Example of common subexpression elimination. . . . .	26
2.6	The middleend of the microC compiler. . . . .	27
2.7	A branching in x86IR represented by basic blocks. . . . .	31
2.8	Comparison of stack states before and after calling <i>foo</i> . . . . .	32
2.9	A peephole optimization. . . . .	38
2.10	The backend of the microC compiler. . . . .	38
3.1	UML class diagram of the optimization implementation. . . . .	47
3.2	UML class diagram of the x86IR implementation. . . . .	50
3.3	UML class diagram of the x86 instruction implementation. . . . .	52
3.4	Generated documentation of the class <i>Optimizer</i> . . . . .	59



---

# List of Listings

1.1	Example of iterative and recursive factorial written in microC.	6
1.2	Example of an x86-64 assembly code.	10
1.3	Example of an analysed microC program snippet.	12
2.1	An optimizable program.	23
3.1	For-cycle approach to iterating over non-terminating instructions of a program.	42
3.2	Functional approach to iterating over non-terminating instructions of a program.	42
3.3	The class <code>Frontend</code> .	43
3.4	The trait <code>AnalysisHandlerInterface</code> .	44
3.5	The class <code>Middleend</code> .	48
3.6	Comparison of NASM and GAS syntax.	52
3.7	The trait <code>Backend</code> and its implementation for the x86 backend.	55
3.8	The class <code>Compiler</code> .	56
3.9	Example of a test suite.	57
3.10	Annotation of the class <code>Optimizer</code> .	58
C.1	The abstract syntax of microC.	80
D.1	The abstract syntax of x86IR.	82
E.1	Example implementation of <code>AnalysisHandlerInterface</code> .	87
E.2	Usage of the class <code>Compiler</code> .	88
F.1	Compiled function <code>foo</code> before optimizations.	92
F.2	Compiled function <code>foo</code> after optimizations.	93
F.3	Compiled function <code>bar</code> before optimizations.	94
F.4	Compiled function <code>bar</code> after optimizations.	95
F.5	Compiled function <code>baz</code> before optimizations.	96
F.6	Compiled function <code>baz</code> after optimizations.	97





---

## List of Tables

- 2.1 The rewrite-rules for binary operations *equal* and *greater-than*. . . 25
- 2.2 The rewrite-rules for 2-instruction and 1-instruction sliding window. 39
- 3.1 Size of the compiler codebase without the NI-APR codebase. . . . 56



---

# Introduction

A typical program written by humans often contains many redundant operations [1, Chap. 9.1.1]. The origin of these redundancies may differ; some of them are available at the source level<sup>1</sup> and some of them are the consequence of writing the program in a high-level language. The task of an optimizing compiler is to transform these redundancies, as well as many other imperfections in a program, in a way that will improve<sup>2</sup> the quality of the program—such transformation is called an *optimization*. However, in order for the compiler to be able to perform these optimizations, it needs to know various properties of the program, such as [3, Chap 1.1]:

- Has the value of an expression already been calculated?
- Is the program statement unreachable for every possible program input?
- Does the value of some variable depend on the program input, or is it always the same value?

To answer such questions, the compiler utilizes *static program analyses*, which aim to give information about the possible behaviours of the program without actually running it [3, Introduction]. However, as stated in Rice's theorem [4], these questions are *undecidable* for Turing-complete languages and therefore the answers given by the analyses generally must involve *approximation*. Depending on the type of approximation, we can divide these analyses into two types [3, Chap. 5.8]:

---

<sup>1</sup>E.g., a programmer may find it more convenient to recalculate some result instead of reusing the already calculated value.

<sup>2</sup>Note that the *improvement* does not necessarily mean a faster execution of the compiled program [2, Chap. 8.1]. It can also mean smaller code size (which does not automatically imply a faster execution) or lesser memory footprint of the running program.

- *May analyses*, which give guarantees about what programs *may* do, therefore computing an *over*-approximation. An example of such analysis is the *live variable analysis*, which gives information about what variables *may* be read (i.e., are live) during the program execution.
- *Must analyses*, which describe information about programs that *must* definitely be true, therefore computing an *under*-approximation. An example of such analysis is the *available expression analysis*, which gives information about what expressions *must* have been already computed earlier (i.e., are available) in the program execution.

While the analyses give only approximative guarantees about the programs, they are still very useful not only for the optimization as we already stated (which was the original purpose of the static program analyses), but also, e.g., for finding bugs in programs.

At FIT CTU, the NI-APR course (Selected Methods for Program Analysis) [5] focuses on some of the most classical program analyses. In the first half of the course, students are implementing static analyses of programs written in *microC*—a tiny C-like programming language, yet with some interesting features such as pointers, records, or function values. The problem is that students only implement the analyses and do not see their effects on programs that could have been done by an optimizing compiler. Therefore, the goal of this work is to fill this gap and create an optimizing compiler for *microC*. The compiler will use the analyses developed by the NI-APR students and produce optimized x86-64 assembly. It should serve as an educational material that helps students to understand the usefulness and importance of the analyses taught in the NI-APR course. With the compiler, students will be able to closely observe the effects of static analyses on the compilation and compare the changes and improvements in the programs produced by the compiler. The main emphasis in the implementation is on compatibility with the course codebase written in Scala and since the compiler will be used as an educational material, the implementation should also be easy to follow (clarity is preferred over conciseness), documented, and well tested.

The thesis is structured into four chapters. We open the background chapter with a brief overview of compiler design, followed by an introduction of the *microC* language, the x86-64 assembly, and static analyses used by the compiler. In the design chapter, we discuss the design of each part of the compiler. Then, in the implementation chapter, we go over implementation, testing, and documentation of the compiler. Lastly, in the assessment chapter, we demonstrate the optimization capabilities of the compiler.

---

# Background

We open this chapter with a brief introduction to compiler design. Then we introduce both source and target languages of our compiler. In the last section of this chapter, we provide a brief overview of static program analyses we will be utilizing in our compiler.

## 1.1 Compiler

Most programming languages are designed to be a human-readable way to express all sorts of computations as a sequence of operations. These operations are then expected to be executed on computer processors. However, the machine operations that a processor implements are often at a much lower-level of abstraction than those defined in a programming language [2, Chap. 1.1]. That's why every programming language operation has to be translated into (often a large number of) machine operations before it can be executed.

The tool that performs these translations is called a *compiler*. Simply put, a compiler is a program that reads a program in one language (called *source language*) and translates it into an equivalent program in another language<sup>3</sup> (called *target language*) [1, Chap. 1.1]. This process is shown in Figure 1.1.

---

<sup>3</sup>Note that the language may not necessarily be a machine code—some compilers produce a program written in another human-readable programming language.

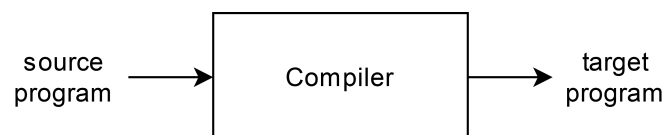


Figure 1.1: A compiler.

There are many types of compilers and compilation strategies. In this thesis, we focus on a classical *ahead-of-time* (AOT) compiler that compiles a high-level programming language into a lower-level one before execution rather than during the execution. Concretely, a compiler that compiles microC source code into Linux native x86-64 assembly. We will properly introduce both of these languages later in this chapter.

### 1.1.1 Modularization

The Figure 1.1 depicts the compiler as a some kind of “black box” that takes the source program and produces the target program. This could incorrectly imply that the compiler takes the source language and maps its functionality directly to the target language, which is often not the case. In reality, the compiler is often decoupled into three smaller “boxes” (as shown in Figure 1.2) which represent three stages of a compilation:

- *Frontend* (or *front end*)
- *Middleend* (or *middle end*)
- *Backend* (or *back end*)

**Frontend** The frontend takes the input source program, parses it and verifies its syntax and semantics. Any syntactic or semantic error gets reported, typically with a source location of the error. The frontend then encodes the input program in some structure for later use by the middleend. This *intermediate representation* (IR) becomes the compiler’s definitive representation for the code it is translating [2, Chap. 1.2]. The IR is usually a lower-level representation of the program compared to the original source.

**Middleend** The middleend performs various analyses (we will discuss some of them at the end of this chapter) on the IR<sup>4</sup> and based on the results of these analyses it performs optimizations that are independent on the target architecture. The middleend can perform multiple of these passes, since some optimizations can enable new optimization possibilities that were not available before. Optimized IR is then passed to the backend.

**Backend** The backend is responsible for mapping the IR passed by the middleend into the target program language (this process is called a *code generation*). Backend usually also performs optimizations that are dependent on the target machine. The output of the backend is a target program.

---

<sup>4</sup>The IR that middleend uses can either be the one provided by the frontend or it can be a different one, generated from the IR provided by the frontend.

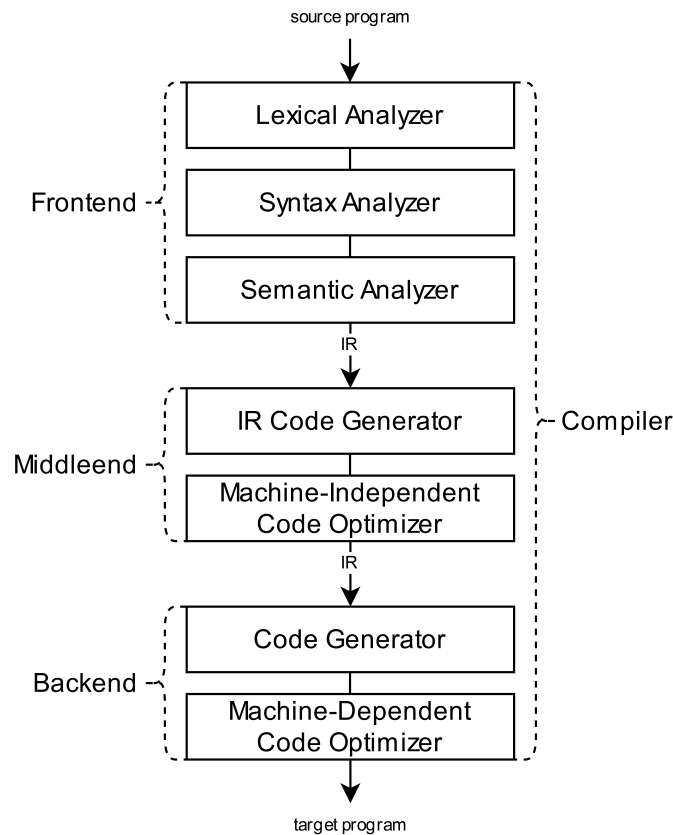


Figure 1.2: The three stages of a compilation (inspired by [1, Fig. 1.6]).

This three-stage modularization makes it possible to combine different frontends and backends while sharing the same middleend optimizations. Some examples of this modularization are the *GNU Compiler Collection*<sup>5</sup> (GCC) or *LLVM*<sup>6</sup>, which both have multiple frontends and backends.

## 1.2 MicroC

The *microC* (stylized as  $\mu C$ ) language is a tiny programming language used in the NI-APR course to implement various program analyses. It is based on the TIP (tiny imperative programming language) introduced in *Static Program Analysis* [3], which is used as the main course book in the NI-APR course. For example of an iterative and recursive factorial program written in microC see Listing 1.1.

<sup>5</sup><https://gcc.gnu.org/>

<sup>6</sup><https://www.llvm.org/>

```
main() {
    var n, f;
    n = input;
    f = 1;

    while (n > 0) {
        f = f * n;
        n = n - 1;
    }

    return f;
}

fac(n) {
    var f;
    if (n == 0) {
        f = 1;
    } else {
        f = n * fac(n - 1);
    }
    return f;
}

main() {
    var n;
    n = input;
    return fac(n);
}
```

Listing 1.1: Example of iterative and recursive factorial written in microC.

In the next few subsections, we take a look at the structure of the microC language. The description of each language feature is taken from NI-APR [6]. Abstract syntax of the microC language is defined in Appendix C. In this thesis, we extend the language with support for arrays, which is not yet (as of 4. 4. 2022) officially part of the course, but planned to be added.

### 1.2.1 Expression

**Identifier** The identifier is any name that starts with an underscore or a letter and contains letters, numbers, and underscores only. The only exception are the following reserved keywords: `alloc`, `else`, `if`, `input`, `null`, `output`, `return`, `var`, and `while`.

**Number** A regular 32-bit signed integer.

**Algebraic operators** There are four basic operators for addition, subtraction, multiplication, and division. Their semantic is defined only for numbers, and their precedence is the same as in maths.

```
x = 2 * 20 + 2;
y = (50 - 8) / 1;
```

**Comparison operators** There are two operators for comparison. The *equality*, which allows comparing numbers and pointers, and the *greater than*



operator, which allows comparing numbers only. Both yield 1 if the comparison is true, 0 otherwise.

```
x = 20 == 10;  
y = 20 > 10;
```

**Input** Reads an integer from standard input. It converts the input into a number or  $-1$  in the case end-of-file (EOF) is reached.

```
x = input;
```

**Function call** A function call consists of one expression, which evaluates to a function value and zero or more expressions, which are the arguments of the function call. Calls use call-by-value semantics and support recursion.

```
x = foo(42);
```

**Pointer** The microC pointers can be recursive (i.e., pointer to a pointer), but no pointer arithmetic is supported. The `alloc` expression allocates an expression on the heap and returns a pointer to it. The `null` expression represents an empty pointer. The dereference operator returns a value that the pointer points to. The reference operator returns a pointer to a variable.

```
x = alloc 42;  
y = *x;  
z = &x;  
x = null;
```

**Record** A record is a value that is composed of one or more fields. Once a record type is assigned to a variable, its fields are fixed and it is not possible to add more fields later. A field can contain any value, except for records, but it can contain a pointer to a record. Records are stack allocated.

```
x = { a: 0, b: 1, c: 2 };
```

**Array** An array is a sequence of fixed length that contains values of the same type. It can contain any value, except for arrays. Arrays are stack allocated.

```
x = [ 0, 1, 2 ];
```

**Comment** The language supports C-style single line and block comments.

```
//single line  
/*  
    multi  
    line  
*/
```

### 1.2.2 Statement

**Assignment** There are five kinds of assignments in microC:

- Direct write—a write to a variable.

```
x = y;
```

- Indirect write—a write to a memory location referenced by a pointer.

```
*x = y;
```

- Direct field write—a write to a record field.

```
x.a = y;
```

- Indirect field write—a write into a field of a record that is referenced by a pointer.

```
(*x).a = y;
```

- Array write—a write to an array element.

```
x[0] = y;
```

**If conditional** In the conditional, only 0 is interpreted as *false*, everything else is interpreted as *true*. The *else* branch of the conditional is optional.

```
if (x) {  
    y = 42;  
} else {  
    y = -42;  
}
```

**While loop** Repeatedly executes its body as long as its guarding condition evaluates to *true* (any non-0 value).

```
while (x) {  
    x = x - 1;  
}
```

**Output** Writes a single integer (i.e., an expression that evaluates to an integer) into the standard output.

```
output 42;
```

**Block** A sequence of other statements (including other blocks) surrounded by curly brackets.

### 1.2.3 Function

Each function has a unique name, takes zero or more arguments, and defines zero or more local variables. All variables are defined at the top of the function. The function has a single return defined at the end of the function body. The function body is a sequence of statements. Functions are also treated as first-class citizens<sup>7</sup>, but the language does not support pointers to functions.

```
main() {  
    var f;  
    f = foo;  
    return f(42);  
}
```

### 1.2.4 Program

A microC program is a simple collection of functions. The entry and exit point of the program is conventionally a function named *main*, which is restricted to take no arguments and return an integer.

## 1.3 x86-64

Intel's *x86-64 ISA*<sup>8</sup> (also known as *Intel 64*) is a 64-bit extension of the original x86 ISA. The x86-64 is a *Complex Instruction Set Computing* (CISC) CPU design, which typically include a wide variety of instructions with varying sizes and wide range of addressing modes [7, Chap. 1.0]. Complete instruction set of x86-64 is available from [8].

As mentioned in the introduction of this thesis, x86-64 will be the target language of our compiler. More specifically, the output of the compiler will be x86-64 assembly code (see Listing 1.2 for an example) executable under any Linux-based 64-bit OS.

### 1.3.1 Storage organization

From our perspective, the executing target program runs in its own address space, in which each program value has a location. The management and organization of this address space is shared between the compiler, operating system, and the target machine [1, Chap. 7.1].

That's why it's important to get more familiar with the storage organization of a program running on the x86 target machine. That way we can get better understanding of, for example, where local variables are stored or how are arguments passed to a function.

---

<sup>7</sup>Values or objects, that can be passed as an argument, returned from a function, and assigned to a variable.

<sup>8</sup>Instruction Set Architecture

```
section .text
global _start
extern exit

_start:
    mov rax, 10    ; store number 10
loop:
    dec rax       ; decrement
    cmp rax, 0    ; compare with number 0
    jne loop      ; jump if not equal
done:
    call exit
```

Listing 1.2: Example of an x86-64 assembly code.

Memory layout of a running x86 program generally consists of a lot of segments, however we will be focusing only on a few of them. Their arrangement is shown in Figure 1.3, which is a simplified version of [9, Fig. 6-1].

**Stack** When a function is called, the machine allocates a chunk of stack memory for it—this chunk is often referred to as a *stack frame* and it contains all the local variables of the function, arguments of the function, and temporary values. These values are accessed using addresses relative to stack pointer<sup>9</sup> (SP) and base pointer<sup>10</sup> (BP) [10, Chap. 12.11]. When the function is returning, the stack frame is deallocated and all of its local variables and arguments become invalid. This allocation and deallocation is a responsibility of the compiler, and it will be discussed in the following chapter.

**Heap** The area for dynamically allocated memory is called the heap. One of the key differences between stack and heap memory is the responsibility for allocation and deallocation. Heap memory is allocated explicitly by a programmer and variables allocated on it never fall out of scope like local variables on the stack. To allocate heap memory in C++ we use keyword `new`. In microC, the keyword `alloc` is used.

**Data** This section stores global and static variables. It is further divided into initialized and uninitialized data segments. Since in microC functions are first-class citizens and they are treated like ordinary variables, this is where they will be stored (since they are accessible from any other scope, i.e., they are global).

---

<sup>9</sup>*Stack pointer* is a pointer that points to the current top of the stack [7, Chap. 2.3.1.2].

<sup>10</sup>*Base pointer* points to the start of a function's stack frame [7, Chap. 2.3.1.3].

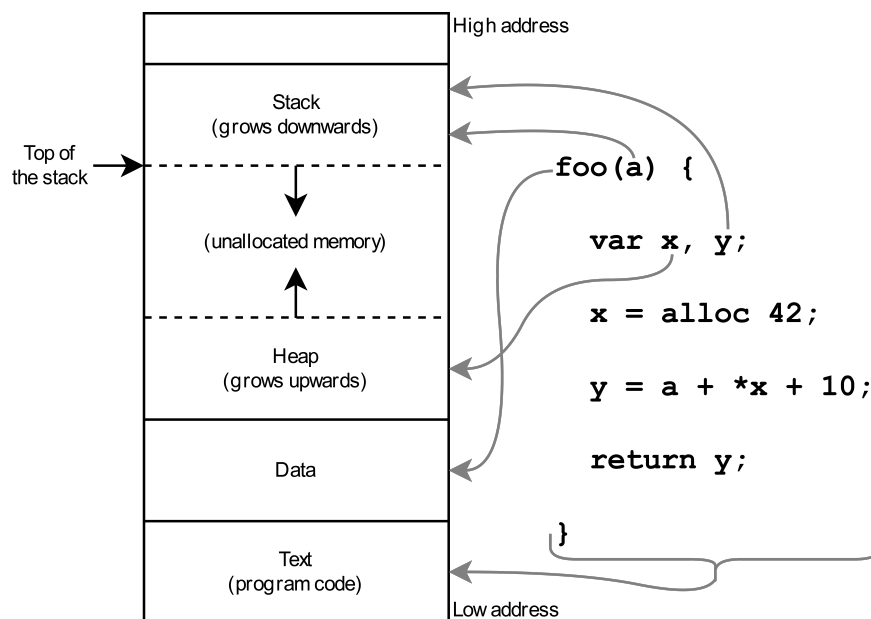


Figure 1.3: Typical memory layout of a running x86 program.

**Text** Text segment is a memory section that contains the code that is being executed. In order to prevent a program from accidentally modifying its instruction, this section is read-only.

## 1.4 Static program analysis

*Static program analyses* are used to reason about the behaviour of computer programs without actually running them [3, Preface]. This is useful for many purposes, such as:

- Program optimization—e.g., detecting unused and unreachable code which can be deleted by the optimizer, detecting variables whose value can be computed at compile time, deciding if two pointers point to the same data structure, or determining bounds of an integer variable.
- Program correctness—e.g., deciding if the program is typeable<sup>11</sup>, detecting use of an undeclared variable, checking if all variables are initialized before they are read, or determining if the program terminates on every possible input.
- Program development—supporting various tools of modern IDEs like debugging, refactoring, code completion, type inference, or detecting where a function can be called from.

<sup>11</sup>A program is *typeable* if it satisfies a collection of type constraints derived from the program [3, Chap. 3].

```
var x, y;
x = 5;
y = {a: x * -10};
if (input) {
    x = x + 1;
} else {
    x = x * -10;
}
y = {a: x};
```

Listing 1.3: Example of an analysed microC program snippet.

In this section, we will get more familiar with some of the analyses used for the first two listed purposes, as those will be utilized in our compiler. The following subsections will use Listing 1.3 as an example of an analysed microC program snippet.

### 1.4.1 Semantic analysis

The *semantic analysis* is an example of an analysis for program correctness. The version that we will use in our compiler checks the following:

- Use of an undeclared identifier.
- Duplicate identifiers.
- Duplicate record field names.
- Assignment to a function.
- Taking an address of a function.

The output of semantic analysis is a mapping of identifiers to their declaration in the program.

### 1.4.2 Type analysis

The *type analysis* is another example of an analysis for program correctness. Its goal is to assign type to each value of a program and decide if the program is typeable. This will prevent the compiler to even process ill-typed programs. The output of type analysis is a mapping of variable declarations to their respective types. Example of a type analysis output for the program from Listing 1.3:

$$\begin{aligned} \llbracket x \rrbracket &= int \\ \llbracket y \rrbracket &= \{a : int\} \end{aligned}$$

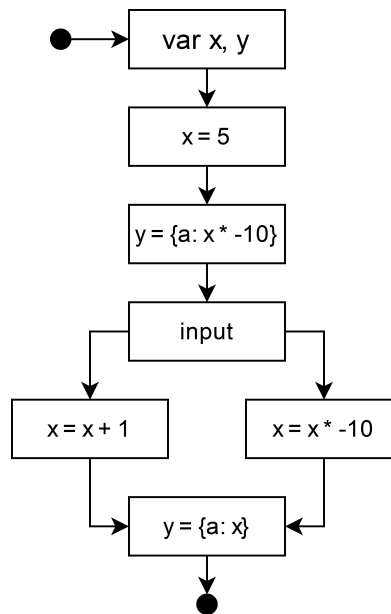


Figure 1.4: CFG of the program snippet from Listing 1.3.

### 1.4.3 Control flow graph

Each of the so far mentioned analyses works with a representation of a program without the need to know the actual dataflow of the program. However, this is not the case for all the analyses we will be talking about. Some of the analyses (which are referred to as *dataflow analyses*) require the knowledge of program's dataflow. For this purpose it is more convenient to work with a different representation of the program called *control flow graph*.

A control flow graph (CFG) is a directed graph, in which *nodes* correspond to statements and *edges* represent possible flow of control [3, Chap. 2.5]. A CFG of a microC program always has a single point of entry (denoted *entry*) and a single point of exit (denoted *exit*). An example of a CFG is shown in Figure 1.4.

### 1.4.4 Sign analysis

The *sign analysis* is an example of a dataflow analysis for program optimization. Its goal is to determine what sign (+, -, or 0) each expression has, which the compiler can then use to simplify certain binary operations<sup>12</sup>. The output of sign analysis is a mapping of CFG nodes to maps from declaration to sign or abstract values  $\top$  (representing “don't know”, i.e., the sign of the declaration

<sup>12</sup>E.g., a binary operation *equals to* between positive and negative integer will always evaluate to false.

is uncertain) and  $\perp$  (representing uninitialized values)<sup>13</sup>. This output tells us what are the signs of variables *after* the given CFG node. Example output for the program from Listing 1.3 (the “ $\llbracket entry \rrbracket$ ” notion denotes a CFG node):

$$\begin{aligned}\llbracket entry \rrbracket &= \{x \rightarrow \perp, y \rightarrow \perp\} \\ \llbracket var\ x, y \rrbracket &= \{x \rightarrow \perp, y \rightarrow \perp\} \\ \llbracket x=5 \rrbracket &= \{x \rightarrow +, y \rightarrow \perp\} \\ \llbracket y=\{a:x*-10\} \rrbracket &= \{x \rightarrow +, y \rightarrow \top\} \\ \llbracket input \rrbracket &= \{x \rightarrow +, y \rightarrow \top\} \\ \llbracket x=x+1 \rrbracket &= \{x \rightarrow +, y \rightarrow \top\} \\ \llbracket x=x*-10 \rrbracket &= \{x \rightarrow -, y \rightarrow \top\} \\ \llbracket y=\{a:x\} \rrbracket &= \{x \rightarrow \top, y \rightarrow \top\} \\ \llbracket exit \rrbracket &= \{x \rightarrow \top, y \rightarrow \top\}\end{aligned}$$

We can see from the output that after the CFG node  $\llbracket y=\{a:x\} \rrbracket$  the sign of the variable  $x$  is uncertain, because it could be either  $+$  or  $-$ , based on which integer the expression `input` evaluates to.

#### 1.4.5 Constant propagation analysis

The *constant propagation* analysis is very similar to the sign analysis. The only difference is that the goal is to determine which variables have a constant value (0, 1, etc.). This knowledge can then be used to simplify certain expressions that can be evaluated during compile time. Output of the analysis is a mapping of CFG nodes to maps from declaration to constant value,  $\top$ , or  $\perp$ . This output tells us what are the constant values of variables *after* the given CFG node. Example output for the program from Listing 1.3:

$$\begin{aligned}\llbracket entry \rrbracket &= \{x \rightarrow \perp, y \rightarrow \perp\} \\ \llbracket var\ x, y \rrbracket &= \{x \rightarrow \perp, y \rightarrow \perp\} \\ \llbracket x=5 \rrbracket &= \{x \rightarrow 5, y \rightarrow \perp\} \\ \llbracket y=\{a:x*-10\} \rrbracket &= \{x \rightarrow 5, y \rightarrow \top\} \\ \llbracket input \rrbracket &= \{x \rightarrow 5, y \rightarrow \top\} \\ \llbracket x=x+1 \rrbracket &= \{x \rightarrow 6, y \rightarrow \top\} \\ \llbracket x=x*-10 \rrbracket &= \{x \rightarrow -50, y \rightarrow \top\} \\ \llbracket y=\{a:x\} \rrbracket &= \{x \rightarrow \top, y \rightarrow \top\} \\ \llbracket exit \rrbracket &= \{x \rightarrow \top, y \rightarrow \top\}\end{aligned}$$

---

<sup>13</sup>More details about these abstract values can be found in [3, Chap. 4], which covers the *Lattice Theory*.



An optimizing compiler will use this output to, for example, propagate the value of the variable  $x$  into the CFG node  $\llbracket y=\{a:x*-10\} \rrbracket$ . The statement of the CFG node will be optimized to  $y=\{a:5*-10\}$ , which can then be further optimized to  $y=\{a:-50\}$ .

### 1.4.6 Live variable analysis

The *live variable analysis* is a dataflow analysis used for program optimization. It calculates which variables are *live* at each point in the program. The compiler can then use this knowledge to optimize variables that are *dead* (i.e., they hold a value that won't be read before the next time they are written to), since they don't need to be stored. The output of the analysis is a mapping of CFG nodes to a set of live variables. This output tells us which variables are live *before* the given CFG node. Example output for the program from Listing 1.3:

$$\begin{aligned} \llbracket entry \rrbracket &= \emptyset \\ \llbracket var\ x,y \rrbracket &= \emptyset \\ \llbracket x=5 \rrbracket &= \emptyset \\ \llbracket y=\{a:x*-10\} \rrbracket &= \{x\} \\ \llbracket input \rrbracket &= \{x\} \\ \llbracket x=x+1 \rrbracket &= \{x\} \\ \llbracket x=x*-10 \rrbracket &= \{x\} \\ \llbracket y=\{a:x\} \rrbracket &= \{x\} \\ \llbracket exit \rrbracket &= \emptyset \end{aligned}$$

We can see that at no point in the program is the variable  $y$  live. An optimizing compiler will use this information to eliminate both assignments to  $y$ , since there is no point in storing dead variables.

### 1.4.7 Available expression analysis

A non-trivial expression<sup>14</sup> is *available* at a program point if its current value has been computed earlier in the execution [3, Chap. 5.5]. The goal of the *available expression analysis* is to determine which expressions are available at the given program point. Output of the analysis is a mapping of CFG nodes to sets of available expressions. This output tells us which expressions are available *after* the given CFG node. Example output for the program from Listing 1.3:

<sup>14</sup>In microC it's the binary operation.

## 1. BACKGROUND

---

$$\begin{aligned} \llbracket entry \rrbracket &= \emptyset \\ \llbracket var\ x, y \rrbracket &= \emptyset \\ \llbracket x=5 \rrbracket &= \emptyset \\ \llbracket y=\{a:x*-10\} \rrbracket &= \{(x*-10)\} \\ \llbracket input \rrbracket &= \{(x*-10)\} \\ \llbracket x=x+1 \rrbracket &= \emptyset \\ \llbracket x=x*-10 \rrbracket &= \emptyset \\ \llbracket y=\{a:x\} \rrbracket &= \emptyset \\ \llbracket exit \rrbracket &= \emptyset \end{aligned}$$

Since the value of the variable  $x$  changes in both of the branching CFG nodes  $\llbracket x=x+1 \rrbracket$  and  $\llbracket x=x*-10 \rrbracket$ , the expression  $(x*-10)$  is no longer available after them and cannot be reused.

---

# Design

Armed with knowledge from the previous chapter, we are ready to discuss the design of the microC compiler. Our main goal is to create a compiler, that can be easily plugged with the results of the various static analyses covered in the NI-APR course. These results will be utilized in various optimizations performed by the compiler. In this chapter, we discuss the design of each part of the compiler, mainly middleend and backend.

## 2.1 The frontend

As described in subsection 1.1.1, the responsibility of a compiler's frontend is to parse the input source program and produce an IR that will be used in the compiler's middleend. For this purpose, we can use the parser provided to students in the NI-APR course, which parses the input microC program and produces an *abstract syntax tree*<sup>15</sup> (AST) representation of the program.

The advantage of this choice is that students of the NI-APR course will already be familiar with the output of the parser. Since there is no need to further modify it, we will use the AST as an output IR of the compiler's frontend (for illustration, see Figure 2.1).

---

<sup>15</sup>*Abstract syntax tree* represents the hierarchical syntactic structure of the source program [1, Chap. 2.1].

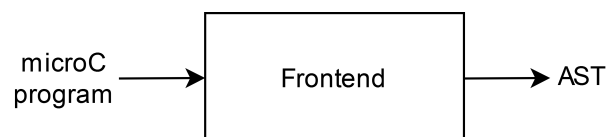


Figure 2.1: The frontend of the microC compiler.

## 2.2 The middleend

The responsibility of our compiler’s middleend will be to utilize provided analyses, use them to perform various kinds of optimizations, and then produce IR from which will the backend generate a target program (x86 assembly). However, this is where we run into our first dilemma—what kind of IR should we use to perform optimizations on?

The idea of *optimizing a program* boils down to rewriting a program so that it computes the same answer in a more efficient way [2, Chap. 1.3.2]. This means that the IR representing a program should be easily modifiable. One option would be to use the IR provided by the frontend—the AST. However, this representation is not well-suited for modifications (since it was not designed for this purpose in the first place). That is why we will go with the second logical option—defining a new IR, that will be well-suited for modifications performed by the optimizer. We will discuss the design of such IR in the following subsection.

### 2.2.1 CFG as an IR

In the subsection 1.4.3 we introduced a notion of a CFG. This representation of a program will be used in most of the analyses that will be provided to our compiler. Coincidentally, it is also quite suitable for performing modifications on it. For example, removal of a program statement can be done by removing the corresponding CFG node and reconnecting its edges. Similarly, adding a program statement is done by simply adding a new CFG node. This makes CFG a suitable candidate for our middleend IR.

The advantage of choosing CFG as our middleend IR is that the optimizer will perform optimizations directly on the CFG. The modified CFG can then be immediately used to run another round of analyses and optimizations<sup>16</sup>, which makes the whole process much simpler. Another advantage is that the NI-APR course already provides a class for constructing CFG from AST called `IntraproceduralCfgFactory`. This class will be used as a first component of the middleend and its output (the CFG of a program) will be used in another important component of the middleend—the optimizer, which we will discuss in the following subsection.

However, to be able to use CFG in the optimizer and also as an output of the middleend (which will backend use to generate x86 assembly), we will have to tackle one issue that CFG has—the loss of information when it comes to branching. Recall the example CFG from Figure 1.4—the if-statement is represented by a branching that starts from the node `input`. However, the context of this branching (which branch is *then* branch and which is *else*) is lost. The same loss of a context applies to while-statements. To make up for this loss, we will have to extend the CFG with a *context* for each of the nodes

---

<sup>16</sup>This “round” of analyses and optimizations is often referred to as an *optimization pass*.

of the CFG. The goal of a *CFG node context* will be to provide additional information about the given CFG node and its branching. There will be four types of contexts in total:

- *Basic Context*—context of a CFG node that is not introducing any form of a branching.
- *If Context*—context of a CFG node that is introducing branching via if-statement (e.g., the input node from the example). It will contain information about which branch is *then* and which is *else*.
- *While Context*—context of a CFG node that is introducing branching via while-statement. It will contain information about which branch is the body of the while-statement and which is the branch that comes after the while-statement.
- *Do-While Context*<sup>17</sup>—context of a CFG node that is the first node of a do-while-statement body. It will contain information about which branch is the body and which is the guard of the do-while-statement.

We will discuss implementation of CFG node contexts in detail in the following chapter.

### 2.2.2 Optimizer

The optimizer is a pivotal component of the compiler’s middleend. Its purpose is to take an IR (in our case CFG), perform various kind of optimizations and produce optimized IR. When designing such optimizer, we have to consider following properties that the optimizer should have:

- interoperability with the provided analyses, and
- extensibility of the optimizer with new optimizations.

The idea is to have an optimizer that can be easily provided with analyses and “plugged with” different kinds of optimizations. The more analyses and optimizations will be provided, the more the optimizer will be effective in optimization of a CFG. This idea is illustrated in Figure 2.2.

The interoperability with the provided analyses can be ensured by defining an interface, that will be used to provide analysis with a CFG and retrieve the result of the analysis. Since this is an implementation detail, we will discuss this interface in the following chapter.

To ensure extensibility of the optimizer with new optimizations, we will have to define a unified approach to performing optimizations. Each of the

---

<sup>17</sup>Even though microC does not define do-while-statements, some optimizations will transform while-statements to do-while-statements. Therefore, it is necessary to define a context for do-while-statement.

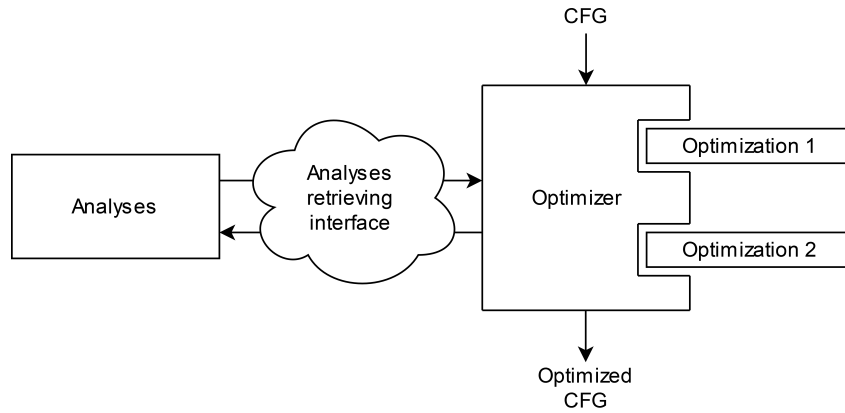


Figure 2.2: An optimizer.

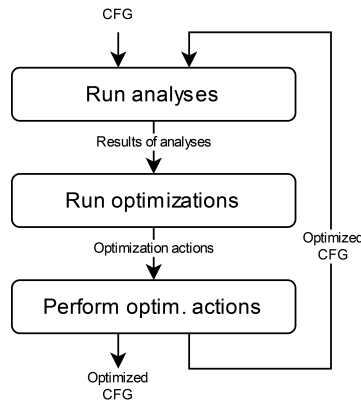


Figure 2.3: An optimization process.

optimizations we will be discussing in the following subsections can be boiled down to a process of observing input CFG, deriving some optimization actions<sup>18</sup>, and performing the optimization actions which will modify the input CFG into an optimized one. The last part of this process (performing optimization actions) is identical for all optimizations, therefore an optimization can be “stripped down” to a process that takes CFG together with results of analyses of the CFG and produces a sequence of optimization actions. This approach simplifies addition of new optimizations, since the one who will be designing a new optimization will not have to be concerned with how the optimization actions will be performed (this will be the concern of the optimizer, which will perform said actions). The only concern will be *how to derive those optimization actions*. For better understanding of the whole process of optimization, refer to Figure 2.3.

<sup>18</sup>E.g., deleting a CFG node, replacing a CFG node with another, introducing a new variable declaration, etc.

So far, we have used the term *optimization action* without giving proper examples and explanations. In the next few paragraphs, we will try to make up for it by covering each of the optimization actions supported by the optimizer.

**Deletion of a node** The node is removed from the CFG while keeping all of its edges. Edges from predecessors of the deleted node are reconnected to successors of the node. Similarly, edges to successors of the deleted node are reconnected to predecessors of the node.

**Connection of nodes** A new directed edge is created between two nodes.

**Disconnection of a node** The node is removed from the CFG together with all of its edges.

**Replacement of a node** The node is replaced with a different node.

**Prependition of a node** The node is prepended with a new node. The new node becomes the only predecessor of the node, and all of its previous predecessors are reconnected to the new node.

**Addition of a declaration** A new declaration of a variable is added.

**Deletion of a declaration** Declaration of a variable is removed.

**Change of a context** Context of the node is changed to a different one.

With examples of optimization actions out of the way, we will now move on to the introduction of optimizations used by the optimizer.

### 2.2.3 Dead statement elimination

The first optimization that will our optimizer use is the *dead statement elimination*. This optimization utilizes the result of the live variable analysis to determine which statements (or, to be precise, which assignments) can be removed without affecting the program results.

Recall the CFG example from Figure 1.4—the variable  $y$  is assigned to in the node  $y = \{a: x * -10\}$ , but the value is never read and is overwritten in the node  $y = \{a: x\}$ . Therefore, since the first assignment does not introduce any side-effect and has no effect on the program, we can safely get rid of it. This constraint is very important, statements that contain side-effect inducing expressions (like `input` or call of a function that contains `input` or `output`) cannot be eliminated, since that would change observable behaviour of the program. In case the Rhs of the assignment contains side-effect inducing

expressions, we will replace the whole assignment with its Rhs. This way we get rid of the assignment while also not changing the observable behaviour of the program.

The live variable analysis tells us which variables are live *before* the given CFG node (cf. subsection 1.4.6). To decide whether the assignment can be eliminated, we have to check if the variable that is being assigned to is dead before every successor of the assignment node. If it is, the node containing the assignment can be safely deleted (or replaced if the assigned expression contains a side-effect).

### 2.2.4 Dead code elimination

The goal of our version of the *dead code elimination* will be eliminating dead branches of *while*, *do-while*, and *if* statements. The branch is dead if it will never be executed for every possible input of the program. There are 4 cases where this happens:

1. If-statement where the guard is a non-zero integer. In this case, the *else* branch together with the guard can be safely eliminated.
2. If-statement where the guard is an integer 0. Here, the *then* branch and the guard can be eliminated.
3. While-statement where the guard is an integer 0. In this case, the body of the while-loop and the guard can be eliminated.
4. Do-while-statement where the guard is an integer 0. Since the body of the loop will be executed only once, the guard can be eliminated and the branch is transformed into a simple sequence of nodes with one entry and one exit.

At first glance it may seem unnecessary to eliminate code that would have never been executed in the first place, however by doing so we are shrinking program size (which is an optimization), but most importantly, this elimination can enable further optimizations. Take the program from Figure 2.4 as an example—by eliminating the *else* branch, we got rid of all reads of the variable *y*, which then enables the dead statement elimination that will eliminate the assignment  $y = 5$ .

To perform this optimization we will simply scan the CFG for guards of *if*, *while*, and *do-while* statements that evaluate to *true* or *false* and then perform the described eliminations by disconnecting entry and exit nodes of the eliminated branch. No analysis is required for this optimization.



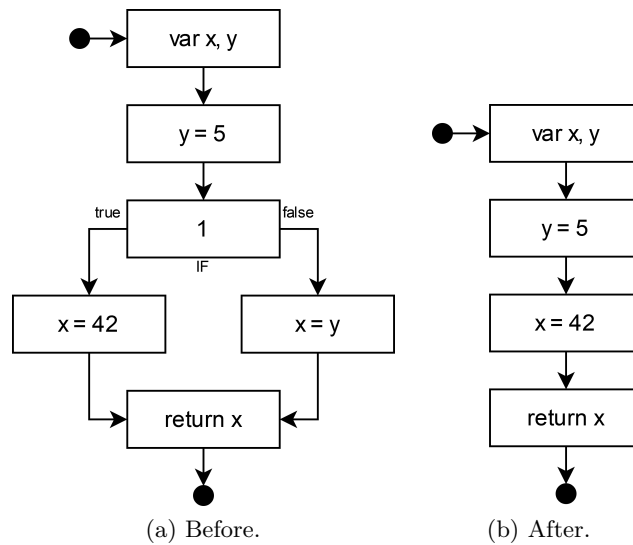


Figure 2.4: Example of dead code elimination.

### 2.2.5 Constant-based dead code elimination

Previous optimization is a simple optimization that does not require any analysis. However, it can be improved by providing more information about the optimized program, e.g., by providing a result of the constant propagation analysis. Take the program from Listing 2.1 as an example—after the first if-statement, the variable `x` will always evaluate to `true` (since both `5` and `10` are interpreted as `true`), therefore both the *else* branch and the guard of the second if-statement can be safely eliminated.

```

if (input) { x = 5; } else { x = 10; }

if (x) { output 1; } else { output 0; }

```

Listing 2.1: An optimizable program.

The *constant-based dead code elimination* is an improved version of the dead code elimination. It utilizes a result of the constant propagation analysis to decide which guards of *if*, *while*, and *do-while* evaluate to `true` or `false` and then performs eliminations described in subsection 2.2.4. In addition, it also performs optimization in the case where the guard of while-statement evaluates to `true` during the first evaluation of the guard. This means that the body of the loop will get executed at least once. In this case, the while-statement can be transformed into a do-while-statement with the same guard and body.

### 2.2.6 Sign-based dead code elimination

The *sign-based dead code elimination* is another example of an improved dead code elimination. It performs the same kind of optimizations as the constant-based dead code elimination, but it utilizes a result of the sign analysis instead.

### 2.2.7 Unused variable elimination

Any declaration of a variable that is never used in the function it was declared in can be safely eliminated. By doing so, we are shrinking the space allocated for local variables on a stack frame (recall storage organization of an x86 target machine discussed in subsection 1.3.1). This optimization is called *unused variable elimination*.

To decide whether a variable is unused, we will scan the CFG for usages of the variable. If no usages are found, we will remove the declaration of the variable. No analysis is required for this scan.

### 2.2.8 Constant folding

The *constant folding* optimization is used to optimize binary operations using a set of rewrite-rules which consist of application of algebraic identities and evaluation of constant expressions. The rules we will be using are listed below (algebraic identities are taken from [1, Chap. 8.5.4]):

$$\begin{aligned}0 + \text{Expr} &\longrightarrow \text{Expr} \\ \text{Expr} + 0 &\longrightarrow \text{Expr} \\ \text{Expr} - 0 &\longrightarrow \text{Expr} \\ \text{Id} - \text{Id} &\longrightarrow 0 \\ \text{Id} * 0 &\longrightarrow 0 \\ 0 * \text{Id} &\longrightarrow 0 \\ \text{Expr} * 1 &\longrightarrow \text{Expr} \\ 1 * \text{Expr} &\longrightarrow \text{Expr} \\ \text{Expr} / 1 &\longrightarrow \text{Expr} \\ \text{Int op Int} &\longrightarrow \text{eval}(\text{Int op Int})\end{aligned}$$

It may seem like some rules for division (e.g.,  $0 / \text{Expr} \longrightarrow 0$ ) are missing, however those cannot be eliminated because of possible division by zero. By rewriting such division, we would change observable behaviour of the program. The same applies to rules such as  $0 * \text{Expr} \longrightarrow 0$ , since the *Expr* itself can contain division by zero.

We will perform this optimization by scanning the CFG for nodes containing binary operations which match the listed rewrite-rules. The nodes are then replaced based on the given rewrite-rules.

==	+	0	-
+	-	false	false
0	false	true	false
-	false	false	-

>	+	0	-
+	-	true	true
0	false	false	true
-	false	false	-

Table 2.1: The rewrite-rules for binary operations *equal* and *greater-than*. Rows and columns represent Lhs and Rhs operands, respectively.

### 2.2.9 Sign folding

The *sign folding* is very similar to constant folding, the only difference is that the rewrite-rules of sign folding utilize the result of the sign analysis. The optimization is applied on binary operations *equal* and *greater-than*, and the rewrite-rules for each of them are listed in Table 2.1. Note that in microC true and false are represented by 1 and 0, respectively. Optimization is also applied to identifiers with a sign 0, which are replaced with a constant 0.

Similarly to constant folding, the optimization is performed by scanning the CFG for nodes containing binary operations and using the result of sign analysis to determine signs of operands of the binary operation. If signs match the listed rewrite-rules, the node is then replaced based on the given rule.

### 2.2.10 Constant propagation

The goal of the *constant propagation* is to substitute variables with constant values (assuming that the constant values are known at compile time) using the result of the constant propagation analysis. This leads to a more optimal program while also possibly enabling further optimizations like constant folding. Recall the CFG example from Figure 1.4—if we use the result of a constant propagation analysis from subsection 1.4.5, we can safely replace the variable  $x$  in the node  $y = \{a : x * 10\}$  with a constant value 5.

The constant propagation analysis tells us constant values of variables *after* the given CFG node (recall subsection 1.4.5). To decide whether a variable in a CFG node can be replaced, we have to check if the variable has constant value in the predecessors of the node. If it has, we can replace every occurrence of the variable in the node with that constant value.

### 2.2.11 Common subexpression elimination

The *common subexpression elimination* gets rid of non-trivial expressions that have been already calculated by the time the flow control reaches them. Results of such expressions are stored in temporary variables, which are then used instead of the redundant expressions [1, Chap. 9.5.1]. This is done by utilizing the result of an available expression analysis. Take the program from Figure 2.5 as an example—expression  $x / 5$  is already available at the return

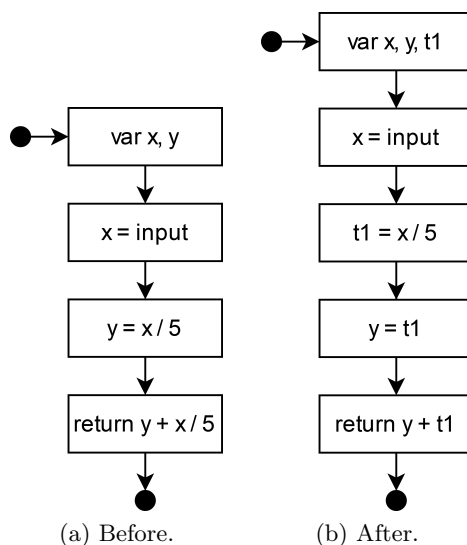


Figure 2.5: Example of common subexpression elimination.

node, therefore we can introduce a new variable `t1` in which we will store this expression and eliminate its every other occurrence.

We will perform this optimization by finding an available expression that is used in the program more than once. Then we will create a new CFG node that stores this expression in a temporary variable and prepend the node to a node that contains the first occurrence of the expression. Every occurrence of the expression (except for the newly created node) can then be replaced with the new temporary variable (assuming that the expression is available at the given point).

### 2.2.12 Summary of the middleend

In this section, we have covered design of every component of the middleend. If we put the components together (as illustrated in Figure 2.6), we will get a middleend that takes an AST from a frontend, constructs a CFG from it, performs various optimizations on the CFG, and produces an optimized CFG together with results of CFG analyses. These results will be further utilized in the backend (e.g., result of type analysis is required for code generation of x86IR, which we will discuss in the following section).

## 2.3 The backend

The responsibility of our compiler's backend will be to use the CFG provided by the middleend to generate an x86 program, which may be further optimized by machine-dependent optimizations. These optimizations typically exploit

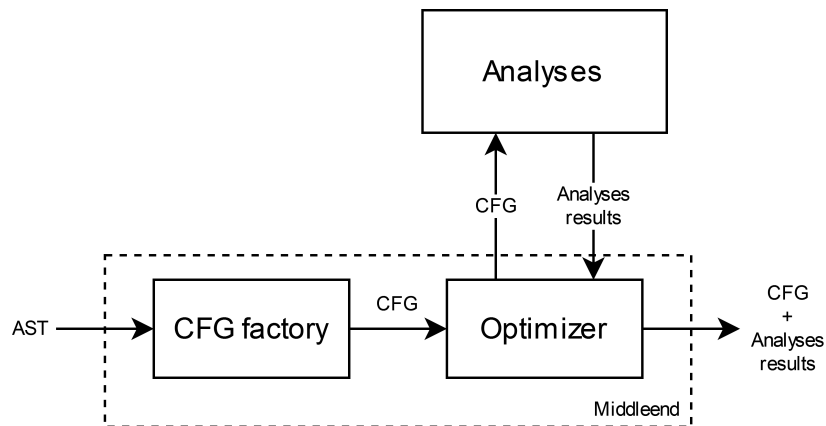


Figure 2.6: The middleend of the microC compiler.

peculiarities of the target architecture [11, Chap. 1.0] (in our case, the x86-64 architecture). In the following subsection, we introduce yet another IR to help us map CFG IR to x86-64 instruction set.

### 2.3.1 x86IR

The subsection 2.2.1 covered CFG IR, which was conveniently designed for various types of analysis and optimizations. One of the nice properties of the CFG IR was the independence from any particular source or target language. However, precisely this independence makes the CFG IR not well suited for a direct compilation into our target language, x86-64 assembly. To tackle this issue, we will have to introduce a lower level IR that will serve as an intermediate step between CFG IR and x86-64 assembly. We will refer to this IR as an *x86IR*.

Generally, an assembly-language program is a form of a linear code. It consists of instructions that execute in their order of appearance [2, Chap. 5.3]. The x86IR is a linear IR that resembles this structure as much as possible. Abstract syntax of the x86IR is defined in Listing D.1.

**Instruction** The x86IR instruction set takes inspiration from the LLVM instruction set [12, Instruction Reference] and consists of a total of 17 instructions (see section D.1 for a detailed overview of the instruction set). These instructions can be divided into 4 categories:

- *Terminator instructions*—these instructions typically yield no value and their sole purpose is to produce control flow at the end of a basic block. Examples: *Return*, *CondJump*, and *Jump*.
- *Memory instructions*—these instructions allow not only to allocate, read and write memory, but also to retrieve or calculate memory addresses

of values. Examples: *Alloc*, *FunAlloc*, *HeapAlloc*, *ArgAddr*, *GetAddr*, *GetAddrOffset*, *Load*, and *Store*.

- *I/O instructions*—instructions that handle I/O operations. Examples: *LoadInput* and *Print*.
- *Other instructions*—instructions that don't fit in any of the previous categories. Examples: *LoadImm*, *LoadComposed*, *BinOp*, and *Call*.

Even though every instruction has different properties and operands, each of them shares common fundamental properties—*location* and *type*.

The location of an instruction is information about its origin in a microC source code (line and column). It is used in error reports to direct the user to the source of the error.

The type of instruction is important information that will come handy for a compilation from x86IR to x86 assembly. It will tell us how to handle the result of an instruction (e.g., how to store it in registers or memory) or the byte-size of the result, which will be used to calculate the total byte-size of a stack frame. There are 4 types in total:

- *VoidType*—type of an instruction that yields no value, therefore the byte-size of the instruction's result is 0.
- *SimpleType*—type of an instruction that returns a non-composed simple value, like an integer or a function. Byte-size of the type is 8 bytes<sup>19</sup> (equal to the size of an x86-64 register).
- *ComposedType*—type of an instruction that returns a value that is composed of one or more values (e.g., *Record*). Its byte-size is a sum of all types of which it consists.
- *PointerType*—type of an instruction that returns an address of some value (equivalent to a pointer in the C programming language). Byte-size of the type is equal to the size of an address (8 bytes).

**Basic block** A *basic block* is a named sequence of instructions that has single entry and single exit point. The exit point (i.e., the last instruction) of a basic block is a terminator instruction (such as *jump* or *function return*). Each basic block contains exactly one terminator instruction.

**Function** Program functions consists of a name of the function, number of parameters, and most importantly a sequence of basic blocks. A function has a single entry point—the first basic block in the basic block sequence.

---

<sup>19</sup>Even though we will be using 32-bit integers, their size on the stack will be 64 bits, since the x86-64 instructions *push* and *pop* allow only 64-bit operands [7, Chap. 9.2][13].

**The x86IR program** Finally, the program in x86IR is represented by a simple list of one or more functions. One of these functions is designated as an entry function (usually the one with the name *main*), which means the function serves as the starting point for program execution.

### 2.3.2 Code generation of x86IR

Now that we are familiar with the x86IR, we can move on to designing a first component of the backend—*CFG Compiler*, which will transform CFG IR into x86IR. MicroC program is in CFG represented as a set of smaller function CFGs. We will try to map each of these function CFGs into an x86IR *function* by compiling each of its nodes. Since the CFG introduced in the NI-APR course defines 3 types of nodes (*CfgFunEntryNode*, *CfgFunExitNode*, and *CfgStmtNode*), in the next few paragraphs we will cover how to compile each of these types.

**CfgFunEntryNode** As the name suggests, this node represents a function entry. We will initialize a new *function* with an empty *basic block*. Then we will emit *ArgAddr* instruction for every parameter of the compiled function. Then we continue by compiling the next successor of the node (there is always only one successor).

**CfgFunExitNode** Once again, the name is self-explanatory—the node represents a function exit. We will finalize the *function* we initialized in the *CfgFunEntryNode* and since this is the last node of the function CFG, the compilation of the function ends here and we move to the next function.

**CfgStmtNode** This node represents a program statement. The way we will compile this node depends on its context (recall subsection 2.2.1), which can be one of the following 4 types: *Basic Context*, *If Context*, *While Context*, and *Do-While Context*.

The most straight-forward is a node with *Basic Context*. Since this node does not introduce any branching, we will simply compile the program statement and then continue by compiling the next successor of the node.

Nodes with *If Context* are a little more complicated, since they introduce branching. First, we will compile the guard of the if-statement. Then we will create 3 empty *basic blocks* which we will refer to as *then*, *else*, and *finally*. After that we will emit *CondJump* instruction which will be based on the evaluated value of the guard jump either to *then* or *else* block. Now we will “enter” the *then* block, compile the entire then-branch, and lastly emit *Jump* instruction that will jump to the *finally* block. The same process is repeated for the *else* block and else-branch. Finally, we will enter the block *finally* and continue with a compilation of a node, that comes first after the whole if-statement. For an illustration of a result of this process, see Figure 2.7a.

Another type of node is a node with *While Context*, which also introduces branching. Similarly to the previous context, we will create 3 empty *basic blocks* which we will refer to as *guard*, *body*, and *finally*. First we will emit *Jump* instruction that will jump to the *guard* block, then enter the *guard* block, compile the guard of the while-statement, and lastly emit *CondJump* instruction which will based on the evaluated value of the guard jump either to *body* or *finally* block. Then we will enter the *body* block, compile the entire body-branch of the while-statement, and lastly emit *Jump* instruction that will jump back to the *guard* block. As a last step, we will enter the block *finally* and continue with a compilation of a node, that comes first after the whole while-statement. For an illustration of a result of this process, see Figure 2.7b.

The last type of node is a node with *Do-While Context*. Again, we will create 3 empty *basic blocks* which we will refer to as *body*, *guard*, and *finally*. First we will emit *Jump* instruction that will jump to the *body* block, then enter the *body* block, compile the body of the do-while-statement, and lastly emit *Jump* instruction that will jump to the *guard* block. Then we enter the *guard* block, compile the guard of the do-while-statement, and lastly emit the *CondJump* instruction which will based on the evaluated value of the guard jump either to *body* or *finally* block. As a last step, we enter the block *finally* and continue with the compilation of a node, that comes first after the whole do-while-statement.

This concludes compilation of a CFG node into x86IR. The compilation of statements and expressions in those nodes is, for the most part, straightforward—first we will compile operands of a statement/expression and then we compile the statement/expression itself. Therefore, it won't be discussed further. The only exception to this is compilation of a field access, which we will cover in the next and last part of this subsection.

**Field access** In a microC code, fields of a record are accessed using field names. However, the x86IR was designed to be more low-level and closer to the x86-64 assembly, therefore it has no concept of “field names”. In memory, a record is stored as a sequence of members (similarly to a struct type in C language [14] or structure type in LLVM [12, Structure Type]), so it only makes sense to access the members of a record by an offset (of bytes) from the address of the record (which will be the first member of the record). The size of the offset depends on the types (and their byte-sizes) of the record's members, which can be inferred by utilizing the result of a type analysis.

Take record  $\{a:0, b:\{c:1, d:2\}\}$  as an example<sup>20</sup>—to access the field `.b`, we will use an offset of 8 bytes from the address of the record (since size of the field `.a` containing an integer is 8 bytes). To access the field `.b.c`, we will use

---

<sup>20</sup>Officially, the microC language does not support nested records [6, Records], however in case this changes in the future, our compiler will support nested fields.



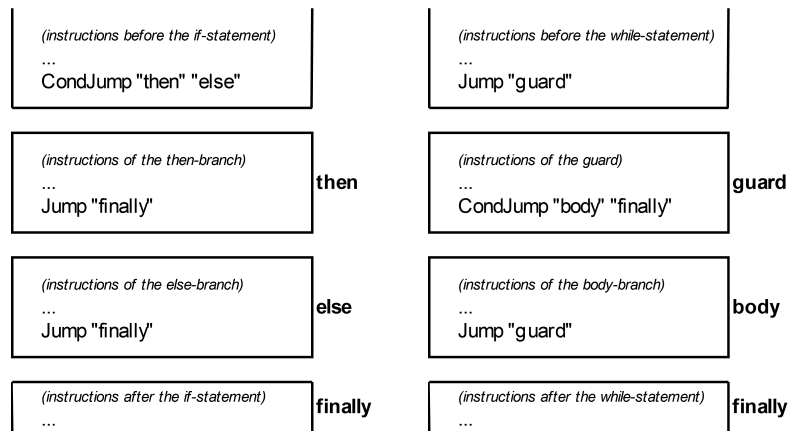


Figure 2.7: A branching in x86IR represented by basic blocks.

an offset of 0 bytes from the address of the record stored in `.b`. For the purpose of addressing by offsets, we will use the x86IR instruction *GetAddrOffset* (see section D.1), which is a simplified version of the instruction *getelementptr* from the LLVM instruction set [12, *getelementptr* Instruction].

### 2.3.3 Stack frame and calling conventions

As mentioned in subsection 1.3.1, allocation and deallocation of a stack frame is a responsibility of the compiler. Before we move onto the generation of x86 assembly code, we will have to design how the allocation and deallocation will work in our compiler. For the design, we will be using a model situation, where function *main* calls function *foo*. The state of the stack before the call is shown in Figure 2.8a.

When *foo* is being called, *main* has to somehow pass function call arguments to *foo*. Some calling conventions may use registers to pass the arguments [10, Chap. 7], however we will be using stack. By doing so, we will implicitly have dedicated space for arguments on stack in case they will have to be spilled<sup>21</sup> from a register during the register allocation. Therefore, *main* has to push all arguments of the call onto the stack. After that, *main* pushes an *instruction pointer*<sup>22</sup> (IP) onto the stack. It will be used to decide which instruction to execute next after *foo* returns from its call. Finally, *foo* is called and the control flow is transferred to it.

<sup>21</sup>*Spilling* occurs when the register allocator cannot assign some virtual register to a physical one [2, Chap. 7.2.3] (e.g., when all of physical registers are occupied)—value of one of the physical registers gets spilled to stack, thus freeing up the physical register.

<sup>22</sup>*Instruction pointer* is a pointer that points to the next instruction that will be executed.

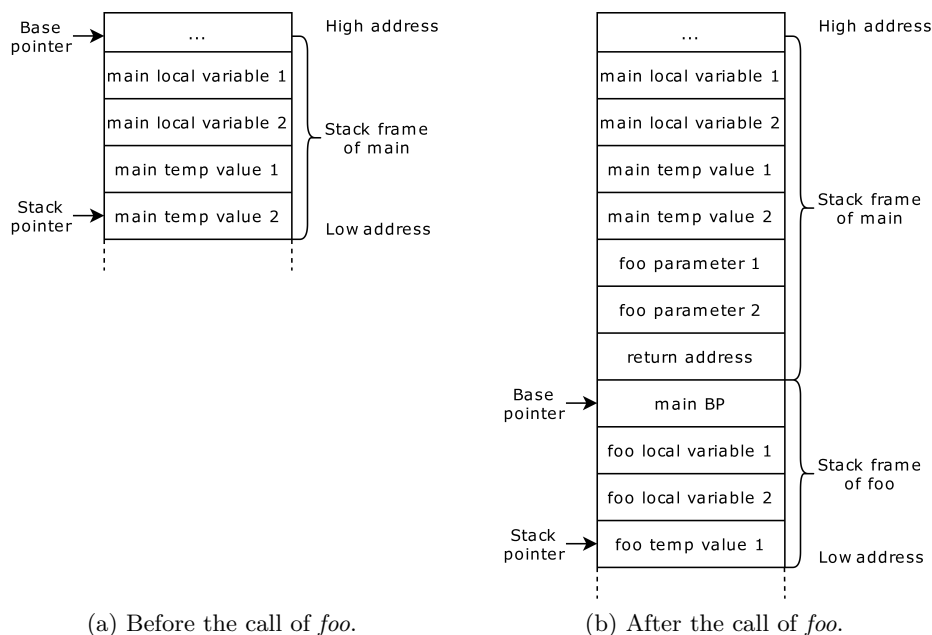


Figure 2.8: Comparison of stack states before and after calling *foo*.

Function *foo* is now in control of the flow and before it executes any of its basic blocks, it has to persist BP of *main* by pushing it onto the stack. Then, *foo* has to set BP to point to the start of the *foo*'s stack frame—this is done by copying SP into BP. Finally, *foo* allocates enough space on the stack for local variables and temporary values of *foo* (this is done by moving the SP downwards) and starts executing its basic blocks. The state of the stack after these actions is shown in Figure 2.8b.

When *foo* reaches a return statement, it has to somehow pass the return value to the stack frame of *main*. Our convention will be to use the register RAX. After the return value is stored, SP is copied into BP and previously saved BP of *main*'s stack frame is loaded back from the stack. Finally, *foo* transfers control flow back to *main* using the IP that was stored by *main* on the stack. At this point, the stack frame of *foo* has been completely deallocated.

Before *main* executes its next instruction, previously pushed arguments of the call are removed from the stack by moving SP upwards. After that, *main* continues to execute its instructions. The state of the stack is now the same as it was before the call (see Figure 2.8a).

### 2.3.4 Code generation of x86 assembly

With better understanding of allocation and deallocation of a stack frame, we can move on to the second component of the backend—*x86IR Compiler*. Its goal will be to transform x86IR into x86 assembly.

Before we start compiling each function of the x86IR, we have to define the initial x86 program entry point with the label `_start`<sup>23</sup>. Then we will have to assign an address to each function of the program (since functions can be referenced and dereferenced in microC). The microC program is then run using the `call` instruction, which will call the entry function of the microC program (usually the one called `main`). After that, the program is terminated by calling `exit` (from `libc`<sup>24</sup>). The other option would be to use `sys_exit` syscall, however since we will use other `libc` functions like `printf` or `scanf`, we will keep things consistent and use `libc` `exit`. We then continue by compiling each function of the program.

**Function** Entry point of a function will be a label with name of the function, which is followed by instructions that take care of updating BP and allocating enough space on the stack for local variables and temporary values (recall subsection 2.3.3 where we discussed what happens, when a function is given control of the flow). After that, we will compile each *basic block* of the function.

**Basic block** Similarly to a function, the entry point of a *basic block* will be a label with its name. Then we will compile each instruction of the *basic block* (their compilation will be discussed in the following paragraphs). After each compiled x86IR instruction we will have to make sure that registers only contain live values—since we only have a finite number of registers, we do not want to keep having registers cluttered with dead values (e.g., operands of an instruction that has been already executed). To decide which values are dead, we will be using a *live instruction analysis*, which will be covered in the following subsection.

**Alloc** The *Alloc* instruction allocates memory on the stack for a local variable. The size of the allocated memory depends on the type of the instruction. Since local variables are addressed using a negative offset from BP (recall Figure 2.8), we will assign a new offset to this variable. The negative offset from BP will be the address of the variable.

**ArgAddr** The *ArgAddr* instruction is very similar to *Alloc*, the only difference is that the memory is allocated for the function's argument and the assigned offset is positive. The positive offset from BP will be the address of the function's argument.

**HeapAlloc** The *HeapAlloc* instruction allocates memory on the heap. To do this, we will have to call the function `malloc` (from `libc`), which takes care

---

<sup>23</sup>This particular label name is recognized by the standard system linkers [7, Chap. 4.6].

<sup>24</sup><https://man7.org/linux/man-pages/man7/libc.7.html>

of the allocation and returns the address of the allocated memory. The call consists of these steps:

1. Save caller saved registers by pushing them onto the stack—these registers are not preserved across a function call. The caller saved registers are: `rax` (since it will contain the return value), `rcx`, `rdx`, `rsi`, `rdi`, `r8`, `r9`, `r10`, and `r11` [7, Chap. 12.8.3].
2. Align stack to 16 bytes—before every call, the stack has to be aligned to 16 bytes [8, Chap 6.2.2].
3. Load the first argument of `malloc` (size of the allocated memory)—following the Linux calling convention, the first argument of a function is stored in the register `rdi` [7, 12.8.1].
4. Finally, call the function.

After the call, the address of the memory will be stored in `rax`. Now we will dealign stack and restore caller saved registers. Lastly, we will initialize the address with the value specified by the *HeapAlloc* instruction.

**Store** The *Store* instruction represents a write to memory, and its arguments are the value to store and the address at which to store it. If the value is a composed type (e.g., a record), the value is actually an address of the composed value we will be storing (its contents are typically stored on stack or heap). We will use this address to access each part of the composed value and move it to the destination address. Any other type is simply moved from a register to the destination address.

**LoadImm** The *LoadImm* instruction loads an immediate value (signed integer). We will do this by simply moving the value into a register. Since microC uses 32-bit integers, we will use only the lowest 32-bits of the 64-bit register<sup>25</sup>.

**LoadComposed** The *LoadComposed* instruction loads a composed value (e.g., a record). Since composed types consist of multiple values, we cannot move them into a register as we did with an immediate value. Instead, we will allocate space on the stack and move each part of the composed type there. Then we will move the address of its first member into a register—this address will represent the “value” of the composed type.

**Load** The *Load* instruction loads a value from a memory address. We will do this by accessing the address and moving its content into a register.

---

<sup>25</sup>This is done by, e.g., accessing the register `rax` as `eax` [7, Chap. 2.3.1.1].

**GetAddr** The *GetAddr* instruction returns the address of a variable. Since the address of a variable is offset from BP, which was assigned to the variable by instruction *Alloc* or *ArgAddr*, we will simply find the assigned offset and load the address into a register.

**GetAddrOffset** The *GetAddrOffset* instruction returns the address of a subelement of a composed type. Arguments of this instruction are an address of the composed type and an offset, therefore we will simply add the offset to the address and load the result of this addition into a register.

**BinOp** The *BinOp* instruction performs a binary operation and returns its result. The most simple are the operations *addition*, *subtraction*, and *multiplication*, which we will perform by emitting x86 instructions `add`, `sub`, and `imul`, respectively.

*Division* is a little more complicated, since the first operand (dividend) of the x86 instruction `idiv` spans over two registers—`rdx` (high bits) and `rax` (low bits) [8, Page 3-487]. We will move dividend into `rax` and set register `rdx` to 0. After executing `idiv`, the result of the operation will be stored in the register `rax`.

In case of *equal* and *greater-than*, we will compare their operands. If the comparison is true, we will load a value 1 into a register. Otherwise, we will load value 0.

**LoadInput** The *LoadInput* instruction loads an integer from a standard input. Similarly to *HeapAlloc*, we will have to use extern function `scanf` (from *libc*), which scans the input using the provided format string. The call of the function consists of these steps:

1. Save caller saved registers by pushing them onto the stack.
2. Align stack to 16 bytes.
3. Load the first argument of `scanf` into the register `rdi`—format string, which defines layout of the scanned input. Since we are loading a 32-bit signed integer, we will use the format string `%ld` (long integer).
4. Load the second argument into the register `rsi`<sup>26</sup>—address of the memory, where the scanned integer will be stored.
5. Finally, call the function.

---

<sup>26</sup>The register `rsi` is used for the second argument of a function in the Linux calling convention [7, 12.8.1].

Right after the call, we will dealign stack. If the return value of the call<sup>27</sup> (stored in `rax`) is 1, we will then move the loaded integer from the address which we provided to `scanf` into a register. Otherwise, EOF was reached and we will load `-1` instead. Lastly, we will restore caller saved registers.

**Print** The *Print* instruction writes an integer to a standard output. Once again, we will have to rely on extern function `printf` (from *libc*), which writes an output using the provided format string. The call of the function consists of these steps:

1. Save caller saved registers by pushing them onto the stack.
2. Align stack to 16 bytes.
3. Load the first argument of `printf` into a register `rdi`—format string, which defines layout of the printed output. We will use the same format string as we did for *LoadInput*—`%ld` (long integer).
4. Load the second argument into a register `rsi`—the value to be printed.
5. Finally, call the function.

After the call, we will just dealign stack and restore caller saved registers.

**Call** The *Call* instruction transfers the control flow to the called function. The call consists of these steps:

1. Save current state of registers by pushing them onto the stack.
2. Align stack to 16 bytes.
3. Load the function that will be called.
4. Load all arguments of the function call—our calling convention is to pass function call arguments using the stack, therefore we will push them in order onto the stack.
5. Call the function using x86 instruction `call`.
6. Clear arguments from the stack.
7. Dealign the stack.
8. Restore previous state of registers from the stack.

---

<sup>27</sup>The function `scanf` returns the number of successfully matched input items [15].

After these steps, the return value will be stored in the register `rax`. In case the return value is a composed type, the value stored in the register `rax` is just its address pointing to a deallocated frame stack of the called function. Therefore, we will have to use this address to move each part of the composed type into the current stack frame.

**Return** The *Return* instruction stores the return value and returns the control flow to the caller of the current function. Since our calling convention is to use register `rax` as the register for return values, we will move the return value into it. Then we will reset `SP` by overwriting it with `BP` and restore `BP` of the caller function by loading it from the stack. Lastly, we will emit x86 instruction `ret`.

**CondJump** The *CondJump* instruction transfers the control flow to a different basic block based on the value of the condition. We will compare the value of the condition with 0 and then emit x86 instruction `je`, which will jump to “false-branch” basic block if the compared values are equal. Then we will emit `jmp` which will jump to “true-branch” basic block.

**Jump** The *Jump* instruction transfers the control flow to a different basic block. We will do this by simply emitting x86 instruction `jmp`.

### 2.3.5 Live instruction analysis

As mentioned earlier, the x86 target machine has only a finite number of registers. Keeping dead values in them will result in a faster congestion of registers, which leads to more spilling (and therefore less efficient code). To decide which values are no longer required to keep in registers, we will be using a live instruction analysis.

The scope of the analysis will be a single basic block  $B$ , which can be viewed as a sequence of  $n$  instructions  $I_0, I_1, \dots, I_n$ . The result of the analysis will be sets of instructions (for each instruction  $I_i \in B$ ), whose result is live *after* the execution of the instruction  $I_i$ , which we will denote as  $LIVE(I_i)$ . The result of the analysis can be defined by the following recurrence relation, where  $op(I_i)$  denotes operands of an instruction  $I_i$ :

$$LIVE(I_n) = \emptyset \tag{2.1}$$

$$LIVE(I_{i-1}) = \left( LIVE(I_i) \cup op(I_i) \right) \setminus \{I_i\} \tag{2.2}$$

No value is live after the last instruction of basic block (relation 2.1). Only values that are yet to be used as operands of some instruction (that has not been executed yet) are live (relation 2.2).

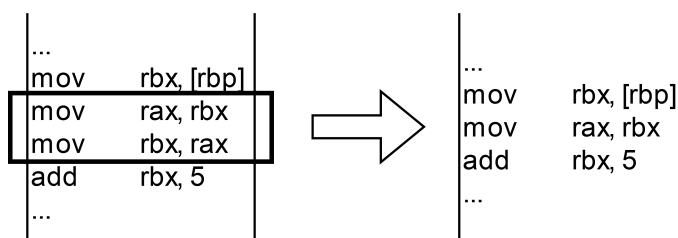


Figure 2.9: A peephole optimization.

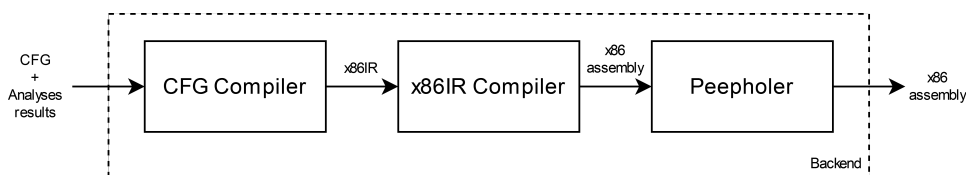


Figure 2.10: The backend of the microC compiler.

### 2.3.6 Peephole optimizations

The last component of the backend will be *Peepholer*. Its goal will be to take assembly produced by *x86IR Compiler*, perform peephole optimizations, and produce optimized assembly.

The basic premise of peephole optimization is to examine short sequences of adjacent operations and find local improvements. This is done by having a sliding window which at each step observes a small number of operations, looking for a specific pattern that can be improved [2, Chap. 11.5.1]. See Figure 2.9 for an example of a sliding window that has the size of two instructions—the first observed instruction copies the content of `rbx` to `rax`. The following instruction then copies the content of `rax` back to `rbx`, which is in this case a redundant operation that can be safely eliminated.

Our peepholer uses sliding window that has size of one and two instructions. The 1-instruction improvements mostly consist of deletion of useless operations, usage of algebraic laws, or usage of instructions designed for special operand cases. The 2-instruction improvements consist of combining of several algebraic operations into one equivalent and eliminating redundant move operations. The overview of all rewrite-rules we will be using is shown in Table 2.2.

### 2.3.7 Summary of the backend

This concludes design of every component of the backend. By putting them all together (as illustrated in Figure 2.10), we get a backend that takes a CFG from a middleend, transforms it into an x86 assembly, performs peephole optimizations on the assembly, and produces an optimized x86 assembly.



2-instruction sliding window		1-instruction sliding window	
Pattern	Replace with	Pattern	Replace with
jmp l1 l1:	l1:	add o1, 0	
add o1, i1 add o1, i2	add o1, $ev(i1 + i2)$	add o1, 1	inc o1
add o1, i1 sub o1, i2	add o1, $ev(i1 - i2)$	add o1, -1	dec o1
sub o1, i1 sub o1, i2	sub o1, $ev(i1 + i2)$	add o1, o1	shl o1, 1
sub o1, i1 add o1, i2	sub o1, $ev(i1 - i2)$	sub o1, 0	
imul o1, i1 imul o1, i2	imul o1, $ev(i1 \cdot i2)$	sub o1, 1	dec o1
shl o1, i1 shl o1, i2	shl o1, $ev(i1 + i2)$	sub o1, -1	inc o1
shr o1, i1 shr o1, i2	shr o1, $ev(i1 + i2)$	sub o1, o1	mov o1, 0
mov o1, o2 mov o2, o1	mov o1, o2	imul o1, 0	mov o1, 0
mov m1, r1 mov r2, m1	mov m1, r1 mov r2, r1	imul o1, 1	
		imul o1, i1 if i1 is power of 2	shl o1, $ev(\log_2(i1))$
		shl o1, 0	
		shr o1, 0	
		mov o1, o1	
		mov o1, 0	xor o1, o1
		mov [o1+0], o2	mov [o1], o2
		mov o1, [o2+0]	mov o1, [o2]

Notes:

l—label                                      r—register  
o—instruction operand                      m—memory access  
i—constant integer                          ev—evaluation of a binary operation

Table 2.2: The rewrite-rules for 2-instruction and 1-instruction sliding window.



---

## Implementation

With the design of the microC compiler out of the way, we can move on to its implementation. In the first part of this chapter we discuss the implementation of each part of the compiler (frontend, middleend, and backend), located in the folder `microc-compiler/src/main/scala/microc`. In the second part, we cover tests and documentation of the implementation.

As stated in the introduction of this thesis, the implementation of the compiler should be easy to read and follow, since it will be used as an educational material for students of the NI-APR course. General ways to achieve that are, e.g., proper commenting and documentation, consistent indentation, following DRY (Don't Repeat Yourself) principle, or using meaningful names for variables and functions. In addition to these principles, we will also take advantage of some useful Scala features; e.g., since Scala is not only an object-oriented programming language, but also a functional programming language [16, Chap. 1.2], we can avoid deep level of nesting (which are often harder to follow) by writing code in a more functional style. Take Listing 3.1 and Listing 3.2 as an example, which are two ways of iterating over non-terminating instructions of an x86IR program—the first one is an imperative for-cycle approach and the second one is a functional approach. The latter one is much easier to follow, since it does not introduce multiple levels of nesting.

On the other hand, there are some Scala features that we will be avoiding for the sake of code clarity. The most notable example of such feature are *implicit conversions*<sup>28</sup>, which are considered (even by the author of Scala, Martin Odersky [17]) “evil”, since they make it hard to see what goes on in the code (e.g., side-effects or complex computations), because they are applied by the Scala compiler on the background.

---

<sup>28</sup> *Implicit conversions* allow the Scala compiler to convert one type to another by utilizing implicit classes or methods defined by the programmer [16, 21.1].

### 3. IMPLEMENTATION

---

```
//functions of a program
for (fun <- program.funs) {
  //basic blocks of a function
  for (bb <- fun.bbs) {
    //instructions of a basic block
    for (i <- bb.instructions) {
      //filter non-terminating instructions
      if (!i.isInstanceOf[Terminator]) {
        println(i.name)
      }
    }
  }
}
```

Listing 3.1: For-cycle approach to iterating over non-terminating instructions of a program.

```
//functions of a program
program.funs
  //basic blocks of a function
  .flatMap(_.bbs)
  //instructions of a basic block
  .flatMap(_.instructions)
  //filter non-terminating instructions
  .filter {
    case _: Terminator => false
    case _              => true
  }
  .foreach(i => println(i.name))
```

Listing 3.2: Functional approach to iterating over non-terminating instructions of a program.

## 3.1 The frontend

We implement the frontend of the microC compiler as an object with single method `parse` which takes a microC program string and produces an AST program (represented by a case class `Program`) by utilizing the class `PCParser` from NI-APR course (as stated in section 2.1). The interface of the object `Frontend` is shown in Listing 3.3 and the implementation is located in `frontend/Frontend.scala`.

```
object Frontend {  
  def parse(code: String): Program = {...}  
}
```

Listing 3.3: The class Frontend.

## 3.2 The middleend

In this section, we go over the implementation of CFG node contexts and the interface between our compiler and the codebase for performing analyses. Next, we discuss the implementation details of the optimizer and the optimization process. Finally, we conclude this section with the introduction of class Middleend, which will encapsulate each component of the middleend. The whole codebase of the middleend is located in the folder middleend/.

### 3.2.1 CFG node context

To preserve the context of branching in CFG, we introduced in subsection 2.2.1 CFG node contexts. In this subsection, we go over the program representation of such contexts and their assignment to CFG nodes.

**Context representation** To be able to pattern match contexts, we will represent them with a sealed trait `CfgNodeContext` extended by four case classes (for each type of the node context): `BasicContext`, `IfContext`, `WhileContext`, and `DoWhileContext`. In the latter three, we will represent a branch with a single CFG node (entry of the branch) and set of CFG nodes (exit of the branch). Each of these representations is located in `cfg/CfgNodeContext.scala`.

**Context assignment** We will need to find a way how to extend the CFG codebase (located in `cfg/Cfg.scala`) provided by the NI-APR course with contexts without changing its interface<sup>29</sup>. One way to achieve this is to add a map that maps CFG nodes to their contexts as an implicit parameter to the class `FragmentCfg`<sup>30</sup>. By doing so, the non-implicit parameter list of `FragmentCfg` will remain the same and the CFG will contain contexts for each of its nodes. The last change is in the class `IntraproceduralCfgFactory` which constructs the CFG of a program. In the method `fromAstNode` which converts AST nodes to CFG nodes, we will need to assign correct contexts at each point where an instance of `FragmentCfg` is created. In the case of AST nodes `IfStmt`

---

<sup>29</sup>Since the CFG codebase is used in the course NI-APR, it has to stay compatible with other codebases used in the course.

<sup>30</sup>A class which represents a fragment of a CFG, which can be a single node, function, or a complete CFG of a program.

and `WhileStmt` we will assign `IfContext` and `WhileContext`, respectively. In all other cases, we will assign `BasicContext`. This small modification will not affect the interface of `IntraproceduralCfgFactory`.

### 3.2.2 Analyses and optimizer interoperability

In the subsection 2.2.2 we discussed a desire to have a compiler that is interoperable with analyses provided to the compiler. This interoperability can be achieved with an interface between our compiler and the codebase for performing analyses. The goal of this interface will be to mediate the process of retrieving the results of analyses.

We introduce trait `AnalysisHandlerInterface` (see Listing 3.4) which defines methods for retrieving the results of each of the analyses we introduced in section 1.4. Analyses will be provided to the compiler by extending the trait and implementing its methods. The only mandatory analyses which will have to be provided are the type analysis and the semantic analysis (those are required by the backend), every other analysis is not required and purely optional. The trait is located in `analysis/AnalysisHandlerInterface.scala`.

```
trait AnalysisHandlerInterface {
  def getTypes(program: Program): Types

  def getDeclarations(program: Program): Declarations

  def getSigns(cfg: ProgramCfg)(implicit declarations: Declarations):
    ⇨ Option[Signs] = None

  def getConstants(cfg: ProgramCfg)(implicit declarations:
    ⇨ Declarations): Option[Constants] = None

  def getLiveVars(cfg: ProgramCfg)(implicit declarations: Declarations):
    ⇨ Option[LiveVars] = None

  def getAvailableExps(cfg: ProgramCfg)(implicit declarations:
    ⇨ Declarations): Option[AvailableExps] = None
}
```

Listing 3.4: The trait `AnalysisHandlerInterface`.

### 3.2.3 Optimizer

With implementation of CFG node contexts and interface for retrieving the results of analyses out of the way, we can finally move on to implementing the pivotal component of the middleend—the optimizer. In Figure 2.3 from the previous chapter, we depicted optimization as a 3-step process that repeats itself until a fixed point is reached. In this subsection, we go over the implementation of each step of this process. The implementation is located in the folder `optimization/`.

**Run analyses** By providing `AnalysisHandlerInterface` we can easily run and retrieve results of the analyses. The results are then stored in a structure called `AnalysesDb` which is a simple case class wrapper that contains each type of analysis result.

**Run optimizations** We will perform each defined optimization in order until of them derives at least one optimization action. In this case, the optimizer performs the derived optimization actions and repeats the whole optimization process from the first step again. If no action was derived, it means that the CFG cannot be further optimized (i.e., the fixed point has been reached). In that case, the whole optimization process ends.

However, there is a small problem we have to address. Since the whole optimization process can repeat itself many times, resulting in too long compilation time, we have to somehow limit the number of repeats of the optimization process. A good compiler must pay attention to compile time costs—this can be achieved by defining a budget for how much time the compiler should spend on its various tasks [2, Chap. 1.3].

We implement this budget by assigning a cost (positive integer) to each of the optimizations and defining a starting budget of the optimizer. Each time an optimization is run, we deduct its cost from the budget. If the budget reaches zero, no more optimizations are run and the optimization process ends.

**Perform optimization actions** First we need to define how to represent these actions. As usual, we define a sealed trait `OptimizationAction` extended by the following case classes representing each of the action types that we introduced in subsection 2.2.2:

- `DeleteNode` with argument `CfgStmtNode` (the deleted node).
- `ConnectNodes` with two `CfgStmtNode` arguments (the source and destination nodes of the new edge).
- `DisconnectNode` with argument `CfgStmtNode` (the disconnected node).

### 3. IMPLEMENTATION

---

- `ReplaceNode` with arguments `AstNode` (the replaced AST node in the CFG node), `AstNode` (the expression replacing the previous one), and `CfgStmtNode` (the node with the replaced expression).
- `PrependNode` with arguments `CfgStmtNode` (the location of the newly created node), `CfgStmtNode` (the new node), and `CfgNodeContext` (the context of the new node).
- `AddDeclaration` with arguments `IdentifierDeclaration` (the declaration of the new variable), `List[Identifier]` (a list of all usages of the new variable), `Type` (the type of the new variable), and `CfgStmtNode` (the node with declarations of all the variables of a function).
- `DeleteDeclaration` with arguments `IdentifierDeclaration` (the deleted declaration) and `CfgStmtNode` (the node with the declaration).
- `ChangeContext` with arguments `CfgNodeContext` (the new context) and `CfgStmtNode` (the affected node).

When an optimization returns a list of optimization actions, the optimizer will pattern match each of the actions and perform corresponding modifications on the CFG.

We encapsulate these three steps into one class `Optimizer`. Its arguments are the CFG that will be optimized, an interface for retrieving the results of analyses, and the budget of the optimizer. The optimizations are performed directly on the provided CFG using the method `run`, which then returns an instance of `AnalysesDb`.

#### 3.2.4 Optimizations

To provide a unified interface for performing optimizations, we introduce a trait `Optimization` (see Figure 3.1). It consists of method `run` which takes a CFG as an input and produces a list of optimization actions, method `isRunnable` which returns `true` if the all necessary analysis required by the optimization are provided, and value `cost` which defines the cost of the optimization. Each optimization we introduced in section 2.2 will be implemented as a class that extends this trait.

Thanks to this unified interface, the optimizer is easily extensible with new optimizations. A new optimization is added to the optimizer by extending the trait `Optimization` and adding it to the list of available optimization in the class `Optimizer` (method `optimizationPlan`).



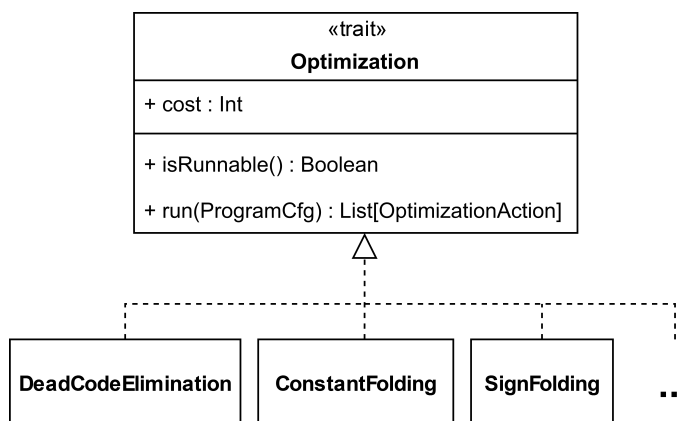


Figure 3.1: UML class diagram of the optimization implementation.

### 3.2.5 Summary of the middleend

In this section, we covered implementation of every component of the middleend. Now we encapsulate these components into a single class `Middleend` (see Listing 3.5) with arguments `AnalysisHandlerInterface` and `Language`. The second argument specifies the used microC language subset:

- `microCVar`: basic microC constructs with no control flow
- `microCIf`: `microCVar` with if-statements
- `microCWhile`: `microCIf` with while-statements
- `microCFun`: `microCWhile` with function calls
- `microCRec`: `microCFun` with records
- `microCArr`: `microCRec` with arrays
- `microC`: entire microC language

The class has a single public method `astToCfg` which scans the input AST and throws an exception, if the AST contains constructs that are not part of the used microC subset. Then the AST is transformed into a CFG and optimized by the optimizer. The used budget of the optimizer depends on the argument `optimize`—if it is set to `true`, the budget is 500. Otherwise, the budget is 0 (meaning no optimization will be performed). The output of the method is a CFG and an instance of `AnalysesDb`.

```
class Middleend(  
  handler: AnalysisHandlerInterface,  
  lang: Language = Language.microC  
) {  
  def astToCfg(  
    program: Program,  
    optimize: Boolean = false  
  ): (ProgramCfg, AnalysesDb) = {...}  
}
```

Listing 3.5: The class Middleend.

### 3.3 The backend

The last part of the compiler we will be implementing is the backend. As stated in subsection 2.3.7, our backend consists of 3 main components: *CFG Compiler*, *x86IR Compiler*, and *Peepholer* (which will be sub-component of the *x86IR Compiler*). In this section, we go over implementation of each of these components (and their subcomponents). The whole codebase of the backend is located in the folder `backend/`.

#### 3.3.1 x86IR

The first part of *x86IR Compiler* is x86IR itself, therefore in this subsection we go over its implementation. The implementation consists of the IR itself and builders of the IR, which are located in `x86IR/X86IR.scala` and `x86IR/X86IRBuilder.scala`, respectively.

**Type of instruction** The only common property of all instruction types is the byte-size, therefore we can model instruction type with a sealed trait `IRType` that defines single method `size`. The trait is then extended by:

- `VoidType`—case object with size 0.
- `PointerType`—case class with size 8 and argument `pointsTo` which defines the type it points to.
- `SimpleType`—case object with size 8.
- `ComposedType`—case class with size defined by the argument of the class.

**Instruction** As mentioned in subsection 2.3.1, the common properties of all instructions are location and type. For the location we can reuse the class `Loc` defined in the codebase of AST and as for the type, we will use

IRType defined earlier. For convenience purposes, we will also define common property *operators*, which will be used to retrieve operands of the instruction in the live instruction analysis. We will model these properties with a sealed trait `Instruction`.

To distinguish terminating instructions from the non-terminating ones, we introduce yet another sealed trait called `Terminator`, which will extend `Instruction`. This trait will have a default type `VoidType` (since terminator instructions yield no value) and instructions `Return`, `CondJump`, and `Jump` will extend it. Every other instruction will extend the trait `Instruction` directly.

Each concrete instruction that directly or indirectly extends the trait `Instruction` is implemented as a case class. By doing so, we will be able to later implement their compilation using a pattern matching mechanism.

**Basic block** A basic block is a named sequence of instructions, which can be represented by a string (name of the basic block) and list of instructions. A basic block should also contain information about the origin of each of its instructions (the source CFG nodes). This information could be useful if, e.g., we wanted to reuse a result of some analysis provided by the middleend (which often maps properties of the program to CFG nodes). For this, we need an association between an instruction and its source CFG node, which can be defined as a map from `Instruction` to `CfgNode`. We model each of these properties with a case class `BasicBlock`.

Since the basic block is created step by step by adding instructions to it, we will use a builder pattern<sup>31</sup> to create it. We define a class `X86BbBuilder` which will have methods for adding instructions and building a basic block from the accumulated instructions. The builder will also prevent invalid actions like adding an instruction to a terminated basic block or building a non-terminated basic block by throwing an exception.

**Function** A function has a similar structure to a basic block, it can be represented by name, list of basic blocks, and number of parameters. Once again, these properties can be defined by a case class `Function`, which will also have methods for extracting local variables and arguments of the function.

When creating a function, we should keep track of already defined variables, so we can assign the correct memory address (represented by `Alloc` instruction) to each reference of a variable. This will be the responsibility of a builder class `X86FunctionBuilder`, which will also have methods for adding basic blocks and building a function from the accumulated basic blocks.

**The x86IR program** Finally, an x86IR program is a simple list of functions, modelled by a case class `ProgramX86IR` which also defines a method

---

<sup>31</sup>*Builder pattern* is used to simplify (often) multistep object creation by defining a class whose purpose is to build instances of another class [18, Chap. 1].

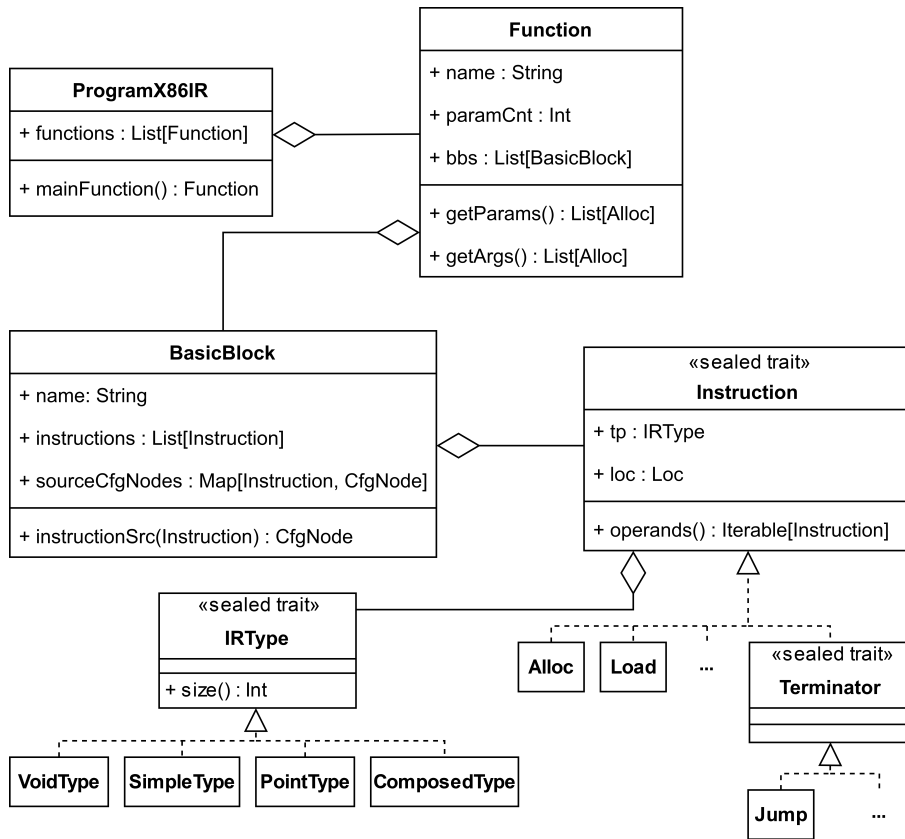


Figure 3.2: UML class diagram of the x86IR implementation.

that returns the entry function of the program. The UML class diagram of the x86IR implementation is shown in Figure 3.2.

We define yet another builder class `X86IRBuilder` which will serve as the main point for building an x86IR program. It will utilize the two builders for building basic blocks and functions we defined earlier on the background. Its main responsibility will be to switch contexts between the basic blocks and functions being built, while also making sure that each name of the created basic blocks is unique and there is no conflict with the names of functions<sup>32</sup>.

### 3.3.2 Code generation of x86IR

With an implementation of the x86IR out of the way, we can finally introduce the class `CfgCompiler` (located in `x86IR/CfgCompiler.scala`). The arguments of the class are CFG of the program and results of type and semantic analysis (both of these are used to derive data types of expressions).

<sup>32</sup>This would result in incorrect behaviour of jump instructions.

The class has a single public method `compile` which follows the compilation rules for each type of the CFG node we defined in subsection 2.3.2 and builds an x86IR program by utilizing the builder class `X86IRBuilder` we defined in the previous subsection. Since the nodes are represented with case classes, we can simply use pattern matching for applying these rules.

In the last part of subsection 2.3.2 we decided to implement field access using an offset from the address of the record. Since the size of this offset depends on the type of record's members, we have to somehow derive their types. For this purpose we define a helper class `TypeAnalyzer` (located in `x86IR/TypeAnalyzer.scala`) which by utilizing the results of type and semantic analysis will derive the correct `IRType` of expressions.

### 3.3.3 x86 assembly

To be able to compile x86IR to x86 assembly, we need to define a program representation of the assembly. The representation should be easily serializable while also allowing to perform peephole optimizations by applying the rewrite rules defined in Table 2.2. In this subsection, we go over such representation. Its implementation, formatter, and builder are located in `x86/helper/`, `x86/X86Formatter`, and `x86/X86Builder.scala`, respectively.

**Instruction representation** When it comes to the shape of the instructions we will be using, we can divide them into 3 types:

1. Zero operand instructions: `name` (e.g., `ret`)
2. One operand instructions: `name op1` (e.g., `inc` or `dec`)
3. Two operand instructions: `name op1, op2` (e.g., `mov` or `add`)

The only common property of these types is the name of the instruction, therefore we will represent an x86 instruction with a sealed trait `Instruction` extended by 3 sealed traits representing each of the shape types, as shown in Figure 3.3. Thanks to this representation, the formatter (which will handle the serialization) will not have to define serialization for each instruction separately, but only for these 3 traits. For convenience purposes, the trait `Instruction` will also allow to optionally assign a comment to an instruction.

To be able to use pattern matching for applying rewrite rules on instructions in the peepholer, we define each concrete instruction as a case class. Similarly, the operands of the instructions (registers, memory accesses, labels, and immediate values) will be represented with case objects and case classes, extending a sealed trait `Operand`.

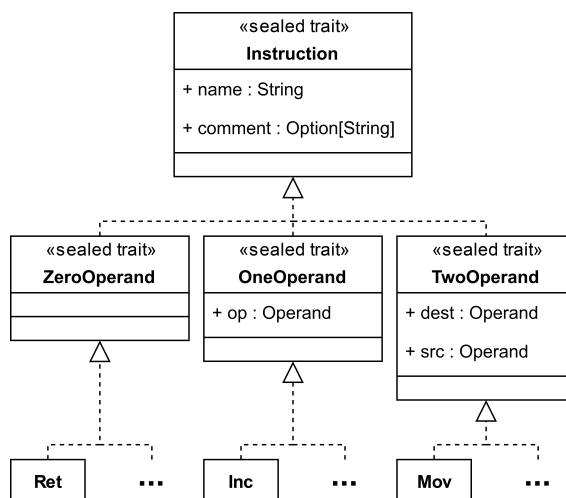


Figure 3.3: UML class diagram of the x86 instruction implementation.

**Formatter** The serialization of the instruction depends on the used syntax of x86 assembly, which often differs in order or format of the instruction operands. An example of these differences is shown in Listing 3.6, which compares NASM<sup>33</sup> syntax and GAS<sup>34</sup> syntax.

<pre> global _start section .text  _start:     mov rax, 5     add rax, 10     ret </pre>	<pre> .global _start .text  _start:     mov \$5, %rax     add \$10, %rax     ret </pre>
------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------

Listing 3.6: Comparison of NASM (on the left) and GAS (on the right) syntax.

To ensure extensibility of supported syntaxes, it is better to define a separate class (or in our case, a trait) for instruction formatting and serialization. Therefore, we introduce a trait `X86Formatter`, which defines methods for formatting instructions and their operands. We also introduce an object `NasmFormatter` which extends the trait and implements formatting of the NASM syntax. The support for another syntaxes is simply added by extending the trait.

<sup>33</sup>Netwide Assembler: <https://www.nasm.us/>

<sup>34</sup>GNU Assembler: <https://www.gnu.org/software/binutils/>

**Builder** We will produce x86 assembly using the builder class `X86Builder`, which accumulates instructions into a list until `build` method is called. Then the builder may apply peephole optimizations to the instruction list by using the `Peepholer` class, which we will discuss in the following subsection. Finally, each instruction is serialized using the formatter which we defined earlier.

### 3.3.4 Peepholer

We implement peepholer as a stateless object `Peepholer` with a single public method `optimize` that repeatedly applies peephole optimizations on a list of instructions until a fixed point is reached. The sliding window can be implemented by a recursive function that observes one or two instructions at a time and applies rewrite-rules from Table 2.2 using a pattern matching<sup>35</sup>. The output of the peepholer will be an optimized list of instructions. The implementation is located in `x86/Peepholer.scala`.

### 3.3.5 Live instruction analysis

The goal of the live instruction analysis is to decide which values are live after execution of the given instruction (recall subsection 2.3.5). We define a class `LiveInstructionAnalysis` which performs the analysis, stores the result, and gives access to the result. The class is located in folder `x86/`.

The class will perform the analysis upon its creation by analysing each basic block of the x86 program. Since the recurrence relation of the analysis (see subsection 2.3.5) is defined from the last instruction of the basic block, we will do this by iterating over the list of instruction in reverse and applying the relation rules. The result of the analysis will be a map that maps each instruction of the basic block to a set of instructions whose value is live. Access to this result will be provided by a method `liveAfterInstr`, which for the given basic block and instruction returns a set of instructions whose value is live after the given instruction.

In addition, the class will also store the order in which the results of instructions will be used as operands of some other instruction. This will be useful information for the register allocation we will be discussing in the following subsection. Access to this order is provided by the method `usageOrder`.

### 3.3.6 Compiler memory

To be able to map x86IR to x86 assembly, we will need to somehow simulate the memory of the x86 target machine, since the compiler needs to keep track of which instruction results are assigned to which registers or which memory

---

<sup>35</sup>This is where the representation of instructions and their operands as case objects and case classes comes in handy, since they come with syntactic conveniences for pattern matching generated by the compiler [16, Chap. 15].

offsets are assigned to which local variables. Another responsibility of the compiler will be to decide which values should reside in registers and which in memory (this process is called a *register allocation* [2, Chap. 11.2]). Therefore, we introduce a class `CompilerMemory` (located in `x86/CompilerMemory.scala`) which will simulate the x86 machine memory and perform a register allocation.

**Memory representation** The function frame stack is represented by two member variables. The first one, `variables`, represents the part of the stack that contains local variables and arguments of the function. It is a map that maps these values to offsets which have been assigned to them by the methods `addLocal` and `addArg`. The second one, `temps`, represents the part of the stack that contains temporary values (e.g., fields of a record). It is a map that maps these values to their offsets assigned by the method `allocateTempOnStack`.

Registers of the x86 machine are represented by the variable `registers`, which maps the registers to values they contain (an instance of `Option`<sup>36</sup>). The values are assigned to registers via the method `allocReg`, which performs the register allocation process.

**Register allocation** The method `allocReg` performs the register allocation for a value by finding an empty register. If no register is empty, it selects a register with a value that will be used furthest in the future and spills it onto the stack. To decide which value will be used furthest in the future, the method uses the class `LiveInstructionAnalysis`, which gives access to the order in which the results of instructions are used.

The method `load` is used to return the register in which the given value is stored. If the value is stored on the stack, it is moved from the stack into an empty register. If no register is empty, the method performs the same spilling process as the method `allocReg`.

#### 3.3.7 Code generation of x86 assembly

The last component implementation we will be discussing in this section is the class `X86IRCompiler` (located in `x86/X86IRCompiler.scala`). The arguments of the class are x86IR program, syntax of the output x86 assembly, and a boolean which determines whether the peephole optimizations should be applied.

The generation of x86 assembly is performed by the method `compile`, which utilizes the classes we discussed earlier (`X86Builder`, `CompilerMemory`, and `LiveInstructionAnalysis`). Since the x86IR is modelled with the usage of case classes, the compilation process we described in subsection 2.3.4 is implemented by using a pattern matching (similarly as in the code generation of x86IR).

---

<sup>36</sup>The class `Option` is a container for zero or one element of a given type. Instances of `Option` are either `Some` (single element) or `None` (no element) [19].



### 3.3.8 Summary of the backend

The last step of backend implementation will be to encapsulate both components `CfgCompiler` and `X86IRCompiler` into a single class representing the backend of the microC compiler. However, before we do that, we will first define a general interface of the microC backend, since in the future more types of backend can be implemented (e.g., a backend that compiles CFG into JVM<sup>37</sup> bytecode). This interface is modelled by a trait `Backend` with a single public method `compile`, which takes `ProgramCfg` (CFG of the program), `AnalysesDb` (results of analyses of the program), boolean `optimize` (determining if the backend should perform optimizations) and returns the target language program representation.

Finally, this interface is implemented by the class `X86Backend`, which puts together and encapsulates both components `CfgCompiler` and `X86IRCompiler`. The class `X86Backend` is shown in Listing 3.7 together with the trait `Backend`.

```

trait Backend[+O] {
  def compile(
    cfg: ProgramCfg,
    analyses: AnalysesDb,
    optimize: Boolean
  ): O
}

class X86Backend(syntax: X86Syntax) extends Backend[String] {
  def compile(
    cfg: ProgramCfg,
    analyses: AnalysesDb,
    optimize: Boolean
  ): String = {...}
}

```

Listing 3.7: The trait `Backend` and its implementation for the x86 backend.

## 3.4 Summary of the compiler

The obvious last step of the compiler implementation is to put all three “pieces of the puzzle” (frontend, middleend, and backend) together. This is done by the class `Compiler` (see Listing 3.8) which encapsulates the whole codebase of the microC compiler. Its usage is described in detail in section E.1.

<sup>37</sup>Java Virtual Machine

```

class Compiler(analysisHandler: AnalysisHandlerInterface, lang:
  ↪ Language, x86Syntax: X86Syntax) {
  def compile(code: String, optimize: Boolean = false): String =
    ↪ {...}
}

```

Listing 3.8: The class Compiler.

This concludes implementation of the microC compiler. The size of the compiler codebase and each of its parts is shown in Table 3.1. Part of the codebase are also example implementations of each analysis covered in section 1.4. For a detailed manual how to use the compiler or how to run programs produced by the compiler, see Appendix E.

	Number of code lines (without empty lines and comments)	Number of classes, traits, and objects
Frontend	7	1
Middleend	1509	54
Backend	1540	116
Whole codebase	3101	175

Table 3.1: Size of the compiler codebase without the NI-APR codebase.

## 3.5 Testing

The project uses two types of tests—unit tests and black-box tests. We will discuss each of these tests in the next two subsections. Both of these tests are also part of a continuous integration managed by GitLab (where the project’s codebase is hosted). With each push or merge request, GitLab runs a pipeline of scripts that will build the project and run both of the mentioned tests. The pipeline is defined in a file `.gitlab-ci.yml` located in the root of the project.

### 3.5.1 Unit tests

Unit testing involves testing software code at its smallest functional point, which is typically a single class [20, Chap. 3.4.5]. The project contains over 90 unit tests (with over 2000 lines of code) which cover every major class discussed in the previous sections of this chapter. Tests are located in `microc-compiler/src/test/scala/microc` and use a library called `uTest`<sup>38</sup>.

Library `uTest` provides a simplistic codebase and functionality for defining test suites and running tests. A test suite is defined by extending abstract

<sup>38</sup><https://github.com/com-lihaoyi/utest>

class `TestSuite` as shown in Listing 3.9, which is an example of a test suite for the class `Peepholer`. Tests can be run from the folder `microc-compiler` via command `sbt test`.

```
object PeepholerTest extends TestSuite {
  val tests: Tests = Tests {
    test("Peepholer test 1") {
      //input
      val instructions = List(
        AddRegImm(RDI(X86RegMode.M64), Imm(1)),
        Jmp(LabelOp("bb1")),
        AddRegImm(RAX(X86RegMode.M64), Imm(5)),
        SubRegImm(RAX(X86RegMode.M64), Imm(8)),
        AddRegImm(RAX(X86RegMode.M64), Imm(3)),
        MovRegReg(RBX(X86RegMode.M64), RBX(X86RegMode.M64)),
        Label("bb1"),
      )

      //assert
      Peepholer.optimize(instructions) ==> List(
        Inc(RDI(X86RegMode.M64)),
        Label("bb1"),
      )
    }
  }
}
```

Listing 3.9: Example of a test suite.

### 3.5.2 Black-box tests

Unlike the unit testing, the black-box testing involves testing of a software code where the inner program structure of the code is not pertinent. The test design is based strictly on the expected program functionality [20, Chap. 8.2.2] (in our case, what output should be produced by the input microC program compiled by our compiler).

The tests are run from the folder `test` via script `./test.sh`. The script compiles and runs every microC program in the folder `test/in` (there are over 30 test programs) and compares their outputs with the reference outputs located in the folder `test/ref`. For each program, both optimized and unoptimized versions are run.

### 3. IMPLEMENTATION

---

```
/**
 * Compiler for microC that produces x86 assembly
 * @param analysisHandler interface that defines methods ...
 * @param lang defines language of the compiler
 * @param x86Syntax syntax of the output assembly
 */
class Compiler(analysisHandler: AnalysisHandlerInterface, lang:
↳ Language, x86Syntax: X86Syntax) {

  /**
   * Compiles given code and returns x86 assembly as a string
   * @param code code to be compiled
   * @param optimize if true, compiler will perform optimizations
   * @return x86 assembly
   */
  def compile(code: String, optimize: Boolean = false): String
}
```

Listing 3.10: Annotation of the class `Optimizer`.

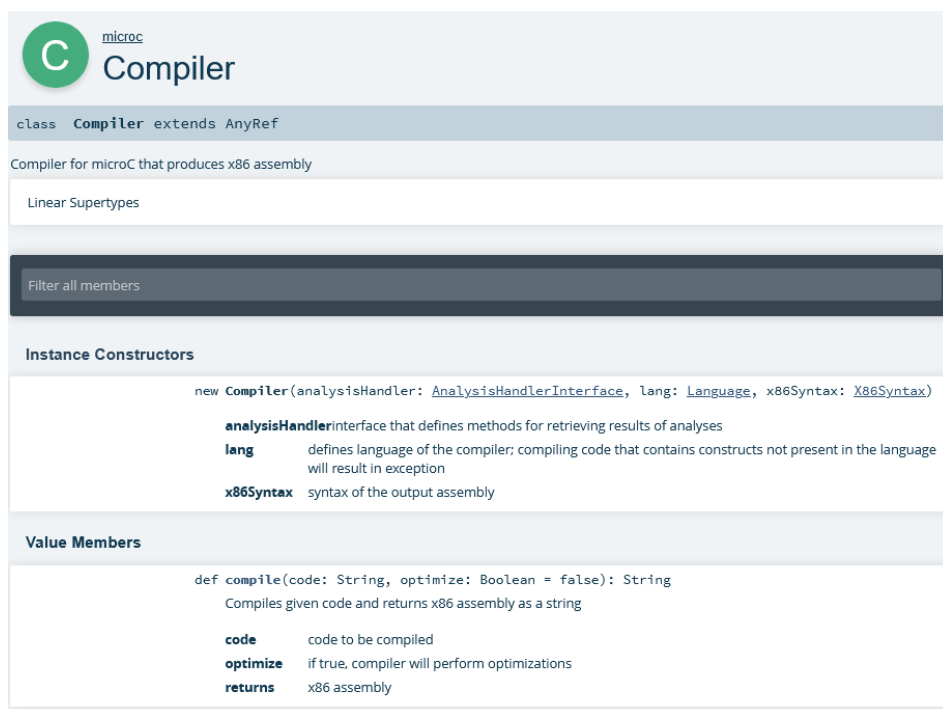
## 3.6 Documentation

Since this project should serve as an educational material, it is important to provide a proper documentation of the project's code-base. The project uses the system *Scaladoc*<sup>39</sup>, which is the most common document generation system for Scala source codes. *Scaladoc* reads specially formatted comments called *annotations* in Scala source code (see Listing 3.10 for an annotated source code example) and generates compiled HTML documentation. *Scaladoc* annotations are also recognized by most of the popular IDEs like IntelliJ IDEA, Visual Studio Code, or Eclipse which offer tools like auto-complete, detection of non-matching annotations, etc.

Documentation can be generated from the folder `microc-compiler` via command `sbt doc`. After running the command, the documentation can be typically found in a folder `microc-compiler/target/scala-X.XX/api`, where "X.XX" depends on the version of Scala. For an example of a generated documentation of the class `Compiler`, see Figure 3.4.

---

<sup>39</sup><https://docs.scala-lang.org/style/scaladoc.html>



The screenshot shows the documentation for the `Compiler` class in the `microC` project. It includes the class signature, a description, linear supertypes, a filter for members, instance constructors, and value members.

**Compiler**  
class `Compiler` extends `AnyRef`

Compiler for microC that produces x86 assembly

Linear Supertypes

Filter all members

**Instance Constructors**

```
new Compiler(analysisHandler: AnalysisHandlerInterface, lang: Language, x86Syntax: X86Syntax)
```

- analysisHandler** interface that defines methods for retrieving results of analyses
- lang** defines language of the compiler; compiling code that contains constructs not present in the language will result in exception
- x86Syntax** syntax of the output assembly

**Value Members**

```
def compile(code: String, optimize: Boolean = false): String
```

Compiles given code and returns x86 assembly as a string

- code** code to be compiled
- optimize** if true, compiler will perform optimizations
- returns** x86 assembly

Figure 3.4: Generated documentation of the class `Optimizer`.



---

# Assessment

In this chapter, we demonstrate the effects of optimizations performed by our compiler by going over the optimization process of three example programs (or, to be precise, functions), step-by-step and compare the produced assembly code before and after optimizations. In the last example, we will also compare the produced assembly code with the one produced by GCC by compiling an equivalent program written in C.

## 4.1 Example 1

Let us consider following function:

```
1 foo(in) {  
2     var x, y, z;  
3     x = 60 * 24;  
4     y = x * in;  
5     if (x > 1400) {  
6         z = y + 10;  
7     } else {  
8         z = y - 10;  
9     }  
10    return z;  
11 }
```

Assembly code generated by compiling this function without any optimization is shown in Listing F.1. At first glance, it is obvious, that this function can be optimized. Without any analysis provided, the compiler will perform the constant folding optimization on the binary operation on line 3:

#### 4. ASSESSMENT

---

```
1 foo(in) {
2   var x, y, z;
3   x = 1440;
4   y = x * in;
5   if (x > 1400) {
6     z = y + 10;
7   } else {
8     z = y - 10;
9   }
10  return z;
11 }
```

At this point, no further optimization can be performed by the compiler without providing more information about the function. One way to do that is to provide the constant propagation analysis to the compiler. The result of this analysis for function `foo` looks like this:

$$\begin{aligned} \llbracket \text{entry} \rrbracket &= \{x \rightarrow \perp, y \rightarrow \perp, z \rightarrow \perp, in \rightarrow \top\} \\ \llbracket \text{var } x, y, z \rrbracket &= \{x \rightarrow \perp, y \rightarrow \perp, z \rightarrow \perp, in \rightarrow \top\} \\ \llbracket x = 1440 \rrbracket &= \{x \rightarrow 1440, y \rightarrow \perp, z \rightarrow \perp, in \rightarrow \top\} \\ \llbracket y = x * in \rrbracket &= \{x \rightarrow 1440, y \rightarrow \top, z \rightarrow \perp, in \rightarrow \top\} \\ \llbracket x > 1400 \rrbracket &= \{x \rightarrow 1440, y \rightarrow \top, z \rightarrow \perp, in \rightarrow \top\} \\ \llbracket z = y + 10 \rrbracket &= \{x \rightarrow 1440, y \rightarrow \top, z \rightarrow \top, in \rightarrow \top\} \\ \llbracket z = y - 10 \rrbracket &= \{x \rightarrow 1440, y \rightarrow \top, z \rightarrow \top, in \rightarrow \top\} \\ \llbracket \text{return } z \rrbracket &= \{x \rightarrow 1440, y \rightarrow \top, z \rightarrow \top, in \rightarrow \top\} \\ \llbracket \text{exit} \rrbracket &= \{x \rightarrow 1440, y \rightarrow \top, z \rightarrow \top, in \rightarrow \top\} \end{aligned}$$

With this knowledge, the compiler can now replace variable `x` on lines 4 and 5 with a constant 1440 by performing the constant propagation optimization:

```
1 foo(in) {
2   var x, y, z;
3   x = 1440;
4   y = 1440 * in;
5   if (1440 > 1400) {
6     z = y + 10;
7   } else {
8     z = y - 10;
9   }
10  return z;
11 }
```



This optimization enables once again the constant folding optimization, which folds the comparison operator on line 5:

```

1 foo(in) {
2   var x, y, z;
3   x = 1440;
4   y = 1440 * in;
5   if (1) {
6     z = y + 10;
7   } else {
8     z = y - 10;
9   }
10  return z;
11 }

```

The next optimization the compiler will perform here is obvious—since the guard of the if-statement on line 5 always evaluates to true, the compiler will eliminate the whole if-statement and replace it with the *then* branch by performing the dead code elimination:

```

1 foo(in) {
2   var x, y, z;
3   x = 1440;
4   y = 1440 * in;
5   z = y + 10;
6   return z;
7 }

```

Once again, the compiler reaches the point where it cannot perform any optimization, not even with the constant propagation analysis, since no variable can be propagated anymore. However, in this situation, the compiler could benefit from a result of the live variable analysis, which gives information about live variables at each point of the program. In case of the function `foo`, the result of the analysis looks like this:

$$\begin{aligned}
\llbracket \text{entry} \rrbracket &= \{in\} \\
\llbracket \text{var } x, y, z \rrbracket &= \{in\} \\
\llbracket x = 1440 \rrbracket &= \{in\} \\
\llbracket y = 1440 * in \rrbracket &= \{in\} \\
\llbracket z = y + 10 \rrbracket &= \{y\} \\
\llbracket \text{return } z \rrbracket &= \{z\} \\
\llbracket \text{exit} \rrbracket &= \emptyset
\end{aligned}$$

As we can see, at no point in the function `foo` is the variable `x` live. The compiler will use this information to eliminate the assignment on line 3 by performing the dead statement elimination:

```
1 foo(in) {
2   var x, y, z;
3   y = 1440 * in;
4   z = y + 10;
5   return z;
6 }
```

Since the variable `x` is no longer used anywhere in the function `foo`, the compiler will perform the unused variable elimination and remove the declaration of variable `x`:

```
1 foo(in) {
2   var y, z;
3   y = 1440 * in;
4   z = y + 10;
5   return z;
6 }
```

At this point, the compiler can no longer perform any optimization. Assembly code generated by compiling the function and performing peephole optimizations is shown in Listing F.2. Compared to the previous assembly code, the number of instructions has been reduced from 40 to 17 and the size of the stack frame has been reduced from 32 to 16 bytes.

## 4.2 Example 2

Let us consider another function:

```
1 bar(in) {
2   var x, y;
3   x = 42;
4   while (x) {
5     if (in) {
6       y = 5;
7     } else {
8       y = 10;
9     }
10    x = 0 / y;
11  }
12  return y;
13 }
```

Assembly code generated by compiling this function without any optimization is shown in Listing F.3. Without any provided analysis, the compiler cannot perform any optimization on this function. Here, the compiler would benefit from a result of the sign analysis, which in case of the function `bar` looks like this:

$$\begin{aligned}
 \llbracket \text{entry} \rrbracket &= \{x \rightarrow \perp, y \rightarrow \perp, in \rightarrow \top\} \\
 \llbracket \text{var } x, y \rrbracket &= \{x \rightarrow \perp, y \rightarrow \perp, in \rightarrow \top\} \\
 \llbracket x = 42 \rrbracket &= \{x \rightarrow +, y \rightarrow \perp, in \rightarrow \top\} \\
 \llbracket x \rrbracket &= \{x \rightarrow \top, y \rightarrow +, in \rightarrow \top\} \\
 \llbracket in \rrbracket &= \{x \rightarrow \top, y \rightarrow +, in \rightarrow \top\} \\
 \llbracket y = 5 \rrbracket &= \{x \rightarrow \top, y \rightarrow +, in \rightarrow \top\} \\
 \llbracket y = 10 \rrbracket &= \{x \rightarrow \top, y \rightarrow +, in \rightarrow \top\} \\
 \llbracket x = 0 / y \rrbracket &= \{x \rightarrow 0, y \rightarrow +, in \rightarrow \top\} \\
 \llbracket \text{return } y \rrbracket &= \{x \rightarrow \top, y \rightarrow +, in \rightarrow \top\} \\
 \llbracket \text{exit} \rrbracket &= \{x \rightarrow \top, y \rightarrow +, in \rightarrow \top\}
 \end{aligned}$$

From this, we can see that during the first evaluation of the loop guard, the variable `x` is always going to be a positive number. And since positive numbers always evaluate to *true*, this means that the while-statement will always execute its body at least once. Therefore, the compiler will utilize the result of the sign analysis to perform the sign-based dead code elimination and transform the while-statement to do-while-statement:

```

1  bar(in) {
2    var x, y;
3    x = 42;
4    do {
5      if (in) {
6        y = 5;
7      } else {
8        y = 10;
9      }
10   x = 0 / y;
11   } while (x);
12   return y;
13 }

```

Another place where the compiler can utilize the result of the sign analysis is the line 10. As follows from the result of the analysis, at that point the variable `y` is always a positive number. Here, the compiler will perform the sign folding optimization, since zero divided by positive number always evaluates to zero:

```
1 bar(in) {
2   var x, y;
3   x = 42;
4   do {
5     if (in) {
6       y = 5;
7     } else {
8       y = 10;
9     }
10    x = 0;
11  } while (x);
12  return y;
13 }
```

The next optimization is obvious—the compiler will use the sign analysis to determine that the guard of do-while-statement always evaluates to zero (i.e., to *false*). Therefore, the compiler will perform the sign-based dead code elimination once again and replace the loop with a body of the loop:

```
1 bar(in) {
2   var x, y;
3   x = 42;
4   if (in) {
5     y = 5;
6   } else {
7     y = 10;
8   }
9   x = 0;
10  return y;
11 }
```

At this point, the compiler cannot perform any optimization, even with the sign analysis provided. This changes if the compiler is provided with a result of the live variable analysis, which for the function `bar` looks like this:

$$\begin{aligned} \llbracket entry \rrbracket &= \{in\} \\ \llbracket var\ x, y, z \rrbracket &= \{in\} \\ \llbracket x = 42 \rrbracket &= \{in\} \\ \llbracket in \rrbracket &= \{in\} \\ \llbracket y = 5 \rrbracket &= \emptyset \\ \llbracket y = 10 \rrbracket &= \emptyset \\ \llbracket x = 0 \rrbracket &= \{y\} \\ \llbracket return\ y \rrbracket &= \{y\} \\ \llbracket exit \rrbracket &= \emptyset \end{aligned}$$

Since the variable `x` is not live at any point of the function `bar`, the compiler will perform the dead statement elimination and eliminate both assignments to the variable `x` on lines 3 and 9:

```

1 bar(in) {
2   var x, y;
3   if (in) {
4     y = 5;
5   } else {
6     y = 10;
7   }
8   return y;
9 }

```

The variable `x` is no longer used anywhere in the function `bar`, therefore the compiler will remove its declaration by performing the unused variable elimination:

```

1 bar(in) {
2   var y;
3   if (in) {
4     y = 5;
5   } else {
6     y = 10;
7   }
8   return y;
9 }

```

This is the last optimization that the compiler can perform. Assembly code generated by compiling the function and performing peephole optimizations is shown in Listing F.4. Thanks to the optimizations, the number of instruction has been reduced from 35 down to 15.

### 4.3 Example 3

Let us consider the last example function:

```
1 baz(in) {
2   var x;
3   if (in * 60) {
4     x = 42;
5   } else {
6     x = 10;
7   }
8   if (x) {
9     x = x + 1;
10  } else {
11    in = in + x;
12  }
13  return 60 * in - x;
14 }
```

Assembly code generated by compiling this function without performing any optimizations is shown in Listing F.5. There are several analyses that can the compiler utilize to improve the function and one of them is the constant propagation analysis<sup>40</sup>:

$$\begin{aligned} \llbracket \text{entry} \rrbracket &= \{x \rightarrow \perp, in \rightarrow \top\} \\ \llbracket \text{var } x \rrbracket &= \{x \rightarrow \perp, in \rightarrow \top\} \\ \llbracket \text{in} * 60 \rrbracket &= \{x \rightarrow \perp, in \rightarrow \top\} \\ \llbracket x = 42 \rrbracket &= \{x \rightarrow 42, in \rightarrow \top\} \\ \llbracket x = 10 \rrbracket &= \{x \rightarrow 10, in \rightarrow \top\} \\ \llbracket x \rrbracket &= \{x \rightarrow \top, in \rightarrow \top\} \\ \llbracket x = x + 1 \rrbracket &= \{x \rightarrow \top, in \rightarrow \top\} \\ \llbracket \text{in} = \text{in} + x \rrbracket &= \{x \rightarrow \top, in \rightarrow \top\} \\ \llbracket \text{return } 60 * \text{in} - x \rrbracket &= \{x \rightarrow \top, in \rightarrow \top\} \\ \llbracket \text{exit} \rrbracket &= \{x \rightarrow \top, in \rightarrow \top\} \end{aligned}$$

We can see that the value of the variable  $x$  after the first if-statement can be either 42 or 10. Both of these values will evaluate on line 8 to *true*, therefore the compiler will perform constant-based dead code elimination and replace the whole second if-statement with its *then* branch:

---

<sup>40</sup>The other is, e.g., the sign-based dead code elimination which would result into a similar optimization to the one we will be discussing.

```

1 baz(in) {
2   var x;
3   if (in * 60) {
4     x = 42;
5   } else {
6     x = 10;
7   }
8   x = x + 1;
9   return 60 * in - x;
10 }

```

That last optimization that the compiler can perform requires the available expression analysis:

$$\begin{aligned}
\llbracket \text{entry} \rrbracket &= \emptyset \\
\llbracket \text{var } x \rrbracket &= \emptyset \\
\llbracket \text{in} * 60 \rrbracket &= \{(\text{in} * 60)\} \\
\llbracket x = 42 \rrbracket &= \{(\text{in} * 60)\} \\
\llbracket x = 10 \rrbracket &= \{(\text{in} * 60)\} \\
\llbracket x = x + 1 \rrbracket &= \{(\text{in} * 60)\} \\
\llbracket \text{return } 60 * \text{in} - x \rrbracket &= \{(\text{in} * 60), (60 * \text{in}), ((60 * \text{in}) - x)\} \\
\llbracket \text{exit} \rrbracket &= \{(\text{in} * 60), (60 * \text{in}), ((60 * \text{in}) - x)\}
\end{aligned}$$

Since the expression  $(\text{in} * 60)$  is available at line 9, and it does not introduce any side-effects, it can be reused. This is done by performing the common subexpression elimination, which will introduce a new variable, that stores the result of  $(\text{in} * 60)$ . Every occurrence of this expression is then replaced by the new variable:

```

1 baz(in) {
2   var x, t0;
3   t0 = in * 60;
4   if (t0) {
5     x = 42;
6   } else {
7     x = 10;
8   }
9   x = x + 1;
10  return t0 - x;
11 }

```

At this point, the compiler can no longer perform any further optimization. Assembly code generated by compiling the function and performing peephole optimizations is shown in Listing F.6. The number of generated instructions has been reduced from 43 down to 29. Since the size of the stack frame is aligned to 16 bytes, the newly introduced variable does not affect it (therefore the size remains 16 bytes).

In the case of this example, it is interesting to compare the output of our compiler with the output of the GCC, which takes a different, more advanced optimization approach. By compiling an equivalent program written in C using GCC with optimization flag `-O2`, we get the following assembly code:

```
baz:
    cmp edi, 1
    sbb edx, edx
    imul eax, edi, 60
    and edx, -32
    add edx, 43
    sub eax, edx
    ret
```

First thing we can notice is that the assembly code does not use the stack. This is because the `baz` function uses only one local variable and there are enough registers to hold all the temporary values during the execution of the program without the need to spill their values to the stack. However, the more interesting thing is that there is no branching. Instead of using conditional jumps like our compiler does, the GCC utilizes a combination of logical and arithmetic operations.

The first instruction compares the parameter `in` with 1, which sets the *carry flag*<sup>41</sup> (CF) to 1 if the parameter `in` is equal to zero, otherwise to 0. The following instruction stores the negative of CF into register `edx`. Then, the instruction `imul` multiplies the parameter `in` with 60 and stores the result into register `eax`. Since the value of register `edx` is either `-1` or `0`, the following instruction `and` will evaluate either to `-32` or `0`, respectively. In other words, if the parameter `in` is zero, the value of register `edx` is `-32`, otherwise it is `0`. This number represents the difference between the possible values of the variable `x` (recall the first `if`-statement of the `baz` function before optimizations). The rest of the assembly code is straight forward—by adding 43, the value of register `edx` now corresponds to the value of variable `x`. The last instruction subtracts the variable `x` from the result of the previous multiplication.

If we compare the assembly code produced by GCC with the one produced by our compiler, it is obvious that there is still a lot of room for improvement in our compiler, such as more advanced instruction selection, or better stack frame allocation.

---

<sup>41</sup> *Carry flag* indicates if the previous operation resulted in a carry [7, Chap. 2.3.1.5].



---

# Conclusion

The goal of this thesis was to develop an AOT optimizing compiler for the microC language that can be plugged with the results of analyses covered in the NI-APR course. This was successfully fulfilled. Not only does the implementation support all features of the language, it also supports arrays, which are (as of 4. 4. 2022) yet to be officially added. The implementation is designed to make it easy to add support for new optimizations or new backends of the compiler. Since the main purpose of the compiler is to serve as an educational material, great emphasis was on clear and easy to follow code, which is also thoroughly documented and well tested.

This work, including the thesis and the codebase, is uploaded on the attached SD card. The work is also available from faculty GitLab in repository: <https://gitlab.fit.cvut.cz/kralva10/microc-optimizing-compiler>

## Future work

Since compiler construction is a complex task, there are many ways in which our work could be further improved.

**Analyses** In our work, we utilized most of the main analyses taught in the NI-APR course. However, there are many more analyses our compiler could utilize and benefit from.

One of such analyses is the *pointer analysis*, which tells us which pointers are *aliases* (can point to the same location) [1, Chap. 12.2.2]. With this information, the compiler can determine what variables are affected by a certain statement and therefore perform more effective register allocation (e.g., by keeping variables in register as long as possible, if they are not affected by any of the program statements).

Another analysis our compiler could utilize is an analysis that detects undefined behaviour in a program. Such behaviour can then be reported to the user during the compilation. For example, the compiler GCC can detect

and report undefined behaviour such as usage of undefined identifier (flag `-Wundef`) or dereference of a null pointer (flag `-Wnull-dereference`) when compiling a program with the corresponding flags [21].

**Optimizations** The most obvious future work improvement is a support of more program optimizations. Their addition should be fairly easy thanks to the unified optimization interface defined in the previous chapter.

One of the most notable classes of optimizations that our compiler could be improved with are the *loop optimizations*, which focus on improving the execution speed and the memory performance of loops. Examples of such optimization are: *loop unrolling*, *loop-invariant code motion*, or *loop fusion*.

Optimizations like *function inlining* or *tail-call optimization* could be used to improve the memory costs of function execution.

**Backends** Thanks to the modularization of the compiler codebase, our work can be quite easily extended with support of more than one backend. For example, the work could be extended with a new backend that compiles the CFG to JVM bytecode by extending the trait `Backend`.

---

# Bibliography

- [1] Aho, A. V.; Lam, M. S.; et al. *Compilers: Principles, Techniques, and Tools*. USA: Addison-Wesley Longman Publishing Co., Inc., second edition, 2006, ISBN 0321486811.
- [2] Torczon, L.; Cooper, K. *Engineering A Compiler*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., second edition, 2007, ISBN 012088478X.
- [3] Møller, A.; Schwartzbach, M. I. *Static Program Analysis*. Department of Computer Science, Aarhus University, November 2020. Available from: <https://cs.au.dk/~amoeller/spa/>
- [4] Rice, H. G. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, volume 74, no. 2, 1953: pp. 358–366, ISSN 00029947.
- [5] NI-APR. Selected Methods for Program Analysis [online]. [visited on 2022-04-23]. Available from: <https://courses.fit.cvut.cz/NI-APR/>
- [6] NI-APR. The microC Programming Language [online]. [visited on 2022-02-25]. Available from: <https://courses.fit.cvut.cz/NI-APR/microc.html>
- [7] Jorgensen, E. *x86-64 Assembly Language Programming with Ubuntu*. Ed Jorgensen, January 2020. Available from: <http://www.egr.unlv.edu/~ed/assembly64.pdf>
- [8] Intel. Intel 64 and IA-32 Architectures Software Developer’s Manual [online]. [visited on 2022-02-18]. Available from: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>
- [9] Kerrisk, M. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. USA: No Starch Press, first edition, 2010, ISBN 1593272200.

- [10] Fog, A. *Calling conventions for different C++ compilers and operating systems*. Technical University of Denmark, February 2010. Available from: [https://www.agner.org/optimize/calling\\_conventions.pdf](https://www.agner.org/optimize/calling_conventions.pdf)
- [11] Seidl, H.; Wilhelm, R.; et al. *Compiler Design: Analysis and Transformation*. Springer Publishing Company, Incorporated, 2016, ISBN 3662507161.
- [12] LLVM. LLVM Language Reference Manual [online]. [visited on 2022-02-03]. Available from: <https://llvm.org/docs/LangRef.html>
- [13] Stack Overflow. 64-bit mode does not support 32-bit PUSH and POP instructions [online]. [visited on 2022-03-24]. Available from: <https://stackoverflow.com/a/43435819>
- [14] cppreference. Struct declaration [online]. [visited on 2022-02-25]. Available from: <https://en.cppreference.com/w/c/language/struct>
- [15] Kerrisk, M. `scanf(3)` — Linux manual page [online]. [visited on 2022-03-14]. Available from: <https://man7.org/linux/man-pages/man3/scanf.3.html>
- [16] Odersky, M.; Spoon, L.; et al. *Programming in Scala: A Comprehensive Step-by-step Guide*. Sunnyvale, CA, USA: Artima Incorporation, first edition, 2008, ISBN 0981531601.
- [17] Scala Contributors. Can We Wean Scala Off Implicit Conversions? [online]. [visited on 2022-04-11]. Available from: <https://contributors.scala-lang.org/t/can-we-wean-scala-off-implicit-conversions/4388>
- [18] Stelting, S. A.; Leeuwen, O. M.-V. *Applied Java Patterns*. Prentice Hall Professional Technical Reference, 2001, ISBN 0130935387.
- [19] Scala Standard Library. Option [online]. [visited on 2022-03-27]. Available from: <https://www.scala-lang.org/api/current/scala/Option.html>
- [20] Huizinga, D.; Kolawa, A. *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Pr, first edition, 2007, ISBN 0470042125.
- [21] GCC. Options to Request or Suppress Warnings [online]. [visited on 2022-04-01]. Available from: <https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>

---

## Acronyms

<b>AOT</b>	Ahead-Of-Time
<b>AST</b>	Abstract Syntax Tree
<b>BP</b>	Base Pointer
<b>CF</b>	Carry Flag
<b>CFG</b>	Control Flow Graph
<b>CISC</b>	Complex Instruction Set Computing
<b>CPU</b>	Central Processing Unit
<b>CTU</b>	Czech Technical University
<b>DRY</b>	Don't Repeat Yourself
<b>EOF</b>	End-Of-File
<b>FIT</b>	Faculty of Information Technology
<b>GAS</b>	GNU Assembler
<b>GCC</b>	GNU Compiler Collection
<b>GNU</b>	GNU's Not Unix
<b>HTML</b>	HyperText Markup Language
<b>IDE</b>	Integrated Development Environment
<b>IP</b>	Instruction Pointer
<b>IR</b>	Intermediate Representation
<b>ISA</b>	Instruction Set Architecture

## A. ACRONYMS

---

**JDK** Java Development Kit

**JVM** Java Virtual Machine

**LHS** Left-Hand Side

**NASM** Netwide Assembler

**OS** Operating System

**PDF** Portable Document Format

**RHS** Right-Hand Side

**SD** Secure Digital

**SP** Stack Pointer

**TIP** Tiny Imperative Programming language

**UML** Unified Modelling Language

---

## Contents of enclosed SD card

doc.....	the directory of documentation
└─ index.html .....	the index file of the documentation
microc-compiler.....	the directory of source files
test .....	the directory of black-box tests
thesis.....	the directory of L <sup>A</sup> T <sub>E</sub> X source codes of the thesis
└─ DP_Král_Václav_2022.pdf.....	the thesis text in PDF format
└─ .gitlab-ci.yml.....	the gitlab CI yml file
└─ microc-compiler.jar.....	the prebuild compiler jar file
└─ README.md.....	the file with project guide





APPENDIX **C**

---

# MicroC

```
Int ::= 0 | 1 | -1 | 2 | -2 | ...
Id  ::= x | y | z | ...
Op   ::= + | - | * | / | > | ==
Exp  ::= Int
      | Id
      | Exp Op Exp
      | ( Exp )
      | Exp ( Exp, ... )
      | null
      | alloc Exp
      | & Id
      | * Exp
      | { Id : Exp, ... }
      | [ Exp, ... ]
      | Exp.Id
      | Exp[ Exp ]
      | input
Stmt ::= id = Exp;
      | * Exp = Exp;
      | Id.Id = Exp;
      | ( *Exp ).Id = Exp;
      | Exp[ Exp ] = Exp;
      | output Exp;
      | { Stmt ... }
      |
      | if ( Exp ) { Stmt } [ else { Stmt } ]
      | while ( Exp ) { Stmt }
Fun  ::= Id (Id, ...) {
      [ var Id, ... ; ]
      Stmt
      return Exp;
      }
Prog ::= Fun ...
```

Listing C.1: The abstract syntax of microC.

APPENDIX **D**

---

# x86IR

```

Int  ::= 0 | 1 | -1 | 2 | -2 | ...
Uint ::= 0 | 1 | 2 | ...
Label ::= x | y | z | ...
Loc   ::= Uint Uint
Op    ::= + | - | * | / | > | ==
Decl  ::= Label Loc
Type  ::= VoidType
       | SimpleType
       | ComposedType Int
       | PointType Type
Instr ::= Alloc Decl Type Loc
       | FunAlloc Decl Type Loc
       | HeapAlloc Instr Type Loc
       | ArgAddr Alloc Type Loc
       | GetAddr Instr Type Loc
       | GetAddrOffset Instr Instr Type Loc
       | Load Instr Type Loc
       | LoadImm Int Type Loc
       | LoadInput Type Loc
       | LoadRecord Instr ... Type Loc
       | Store Instr Instr Type Loc
       | BinOp Op Instr Instr Type Loc
       | Call Instr Instr ... Type Loc
       | Print Instr Type Loc
       | Return Instr Type Loc
       | CondJump Instr Label Label Type Loc
       | Jump Label Type Loc
Block ::= Label Instr ...
Fun    ::= Label Int Block ...
x86IR ::= Fun ...

```

Listing D.1: The abstract syntax of x86IR.

## D.1 Instruction set

**Alloc** The *Alloc* instruction allocates memory on the current stack frame for a local variable of a function and returns its address. The size of the allocated

memory slot depends on the instruction's type. The memory is released when the function returns to its caller, therefore accessing this memory from the caller's stack frame is an undefined behaviour. Operands:

- **decl**: AST identifier declaration of the local variable.
- **type**: *PointerType* that points to the type of the local variable.
- **loc**: Location of the local variable's declaration in a source code.

**FunAlloc** The *FunAlloc* instruction allocates memory in a data segment for a function and returns its address. The address is accessible from every stack frame. Operands:

- **decl**: AST function declaration of the function.
- **type**: *PointerType* that points to function's address type (*PointerType* that points to *SimpleType*).
- **loc**: Location of the function's declaration in a source code.

**HeapAlloc** The *HeapAlloc* instruction allocates memory on a heap, initializes it with the given value and returns its address. Operands:

- **init**: *Instruction* with an initial value of the allocated memory.
- **type**: *PointerType* that points to the type of the initial value.
- **loc**: Location of the allocation in a source code.

**ArgAddr** The *ArgAddr* instruction returns the address of a function's argument. Its sole purpose is to differentiate between arguments and local variables, that are not arguments. I.e., it acts as a wrapper for the *Alloc* instruction. Operands:

- **alloc**: *Alloc* representing the allocated memory of the argument.
- **type**: Same type as **alloc**.
- **loc**: Location of the function's argument declaration.

**GetAddr** The *GetAddr* instruction returns an address of its target argument (equivalent to a reference `&target` in C). Operands:

- **target**: Target *Instruction* whose address is returned.
- **type**: *PointerType* that points to a type of the target.
- **loc**: Location of the reference in a source code.

**GetAddrOffset** The *GetAddrOffset* instruction is used to get an address of a subelement of a composed type. It returns offset address. Operands:

- **addr**: *Instruction* representing an address of a composed type.
- **offset**: *Instruction* representing an offset of the address.
- **type**: *PointerType*.
- **loc**: Location of the address access in a source code.

**Load** The *Load* instruction loads a value from a memory address. Operands:

- **src**: *Instruction* representing an address of a value that will be loaded.
- **type**: Type the type of **src** points to.
- **loc**: Location of the address access in a source code.

**LoadImm** The *LoadImm* instruction loads and returns an immediate integer. Operands:

- **number**: Integer to be loaded.
- **type**: Either *SimpleType* (integer) or *PointerType* (address).
- **loc**: Location of the number in a source code.

**LoadInput** The *LoadInput* instruction loads an integer from a standard input and returns it. If EOF is reached, it returns  $-1$  instead. Operands:

- **type**: *SimpleType*.
- **loc**: Location of the load in a source code.

**LoadComposed** The *LoadComposed* instruction loads a composed value (e.g., a record) and returns a memory address of its first member. Operands:

- **fields**: List of *Instruction* that represent members of the composed value.
- **type**: *ComposedType*.
- **loc**: Location of the composed value in a source code.

**Store** The *Store* instruction writes into a memory address. Operands:

- **dest:** *Instruction* representing a destination address.
- **src:** *Instruction* representing a value to store.
- **type:** *VoidType*.
- **loc:** Location of the assignment in a source code.

**BinOp** The *BinOp* instruction performs a binary operation and returns its result. Operands:

- **op:** Operator of the binary operation. There are 6 defined types:
  - Plus (Addition)
  - Minus (Subtraction)
  - Times (Multiplication)
  - Divide (Division)
  - Gt (Greater than)
  - Eq (Equal to)
- **lhs:** *Instruction* representing a left-hand side of the binary operation.
- **rhs:** *Instruction* representing a right-hand side of the binary operation.
- **type:** Either *SimpleType* (operation on integers) or *PointerType* (operations on addresses).
- **loc:** Location of the binary operation in a source code.

**Call** The *Call* instruction transfers control flow to the called function. After the control flow is transferred back to the caller function, it returns the return value of the function call. Operands:

- **fun:** *Instruction* representing the called function.
- **args:** List of *Instruction* representing arguments of the function.
- **type:** Return type of the function.
- **loc:** Location of the function call in a source code.

**Print** The *Print* instruction writes a value to a standard output. Operands:

- **target:** *Instruction* representing the value to be printed.
- **type:** *VoidType*.
- **loc:** Location of the print call in a source code.

**Return** The *Return* instruction stores the return value and transfers control flow back to the caller of the current function. *Return* is a terminator instruction. Operands:

- **value:** *Instruction* representing the return value of the function.
- **type:** *VoidType*.
- **loc:** Location of the return statement in a source code.

**CondJump** The *CondJump* instruction the transfers control flow to a different basic block in the current function based on the value of the condition. *CondJump* is a terminator instruction. Operands:

- **cond:** *Instruction* representing the condition of the jump. If the condition is evaluated to true, control flow is transferred to `trueTrgt`, otherwise it's transferred to `falseTrgt`.
- **trueTrgt:** Name of the target basic block to which is control flow transferred when the condition is evaluated to true.
- **falseTrgt:** Name of the target basic block to which is control flow transferred when the condition is evaluated to false.
- **type:** *VoidType*.
- **loc:** Location of the control flow transfer in a source code.

**Jump** The *Jump* instruction transfers the control flow to a different basic block in the current function. *Jump* is a terminator instruction. Operands:

- **target:** Name of the target basic block to which is control flow transferred.
- **type:** *VoidType*.
- **loc:** Location of the control flow transfer in a source code.



---

## Usage manual

In this appendix, we show how to use the microC compiler codebase and how to run programs produced by the compiler.

### E.1 How to use

Before running the compiler, the trait `AnalysisHandlerInterface` which defines methods for retrieving analyses results needs to be implemented. Minimum required analyses are type and semantic analyses (which are essential for the compiler), every other analysis is optional and not required. The analyses are provided to the compiler by overriding corresponding methods of the trait. An example of such implementation is shown in Listing E.1, where type, semantic, and sign analyses are provided.

```
object MyAnalysisHandler extends AnalysisHandlerInterface {
  override def getTypes(program: Program): Types = {
    //your implementation of a type analysis (required)
    new TypeAnalysis(program, NoopLogger)(getDeclarations(program)).analyze()
  }
  override def getDeclarations(program: Program): Declarations = {
    //your implementation of a declaration analysis (required)
    new DeclarationAnalysis(program).analyze()
  }
  override def getSigns(cfg: ProgramCfg)(implicit declarations: Declarations):
  ⇨ Option[Signs] = {
    //your implementation of a sign analysis
    val signs = new SignAnalysis(cfg).analyze()
    Some(signs)
  }
}
```

Listing E.1: Example implementation of `AnalysisHandlerInterface`.

Finally, the compiler is run by providing this implementation to an instance of the class `Compiler` and calling the method `compile` with the input microC program, as shown in Listing E.2.

```
val code = ... //the input microC program
val compiler = new Compiler(MyAnalysisHandler, Language.microCVar,
  ↪ X86Syntax.NASM)
val assembly = compiler.compile(code, optimize = true)
```

Listing E.2: Usage of the class `Compiler`.

## E.2 How to run

**Prerequisites** Linux OS is required, since the compiler produces Linux-native x86-64 assembly.

- *nasm*<sup>42</sup>—NASM assembler for assembling of the assembly code produced by the compiler:

```
apt-get install nasm
```

- *ld*<sup>43</sup>—GNU linker required for linking and compiling object files:

```
apt-get install binutils
apt-get install binutils-x86-64-linux-gnu
```

- *libc*<sup>44</sup>—standard C library:

```
apt-get install libc6-dev
```

### Run and compile

1. To run the compiler, you can either (note that the minimum required version of JDK<sup>45</sup> is 11):
  - Use *IntelliJ IDEA* IDE with a Scala plugin<sup>46</sup>.
  - Or use the prebuild jar file located in the root of the project:

```
java -jar microc-compiler.jar MICROC_FILE [-o] >
  ↪ program.asm
```

---

<sup>42</sup><https://linux.die.net/man/1/nasm>

<sup>43</sup><https://man7.org/linux/man-pages/man1/ld.1.html>

<sup>44</sup><https://man7.org/linux/man-pages/man7/libc.7.html>

<sup>45</sup>Java Development Kit

<sup>46</sup><https://www.jetbrains.com/help/idea/discover-intellij-idea-for-scala.html>

where `MICROC_FILE` is the name of file with a microC program and `-o` is an optional flag that decides whether optimizations should be performed or not.

- Or use the building tool *sbt*<sup>47</sup> by running the following command from the folder `microc-compiler/`:

```
sbt run MICROC_FILE [-o] > program.asm
```

2. Assemble the assembly code produced by the compiler:

```
nasm -felf64 program.asm
```

3. Link and compile:

```
ld --dynamic-linker  
↪ /lib/x86_64-linux-gnu/ld-linux-x86-64.so.2 -lc -o  
↪ program program.o
```

4. Run the program:

```
./program
```

---

<sup>47</sup><https://www.scala-sbt.org/>



## **Compiler output examples**

## F. COMPILER OUTPUT EXAMPLES

---

```
foo:
    push rbp
    mov rbp, rsp
    sub rsp, 32
bb_1:
    mov eax, 24
    mov edi, 60
    mov r10d, edi
    imul r10d, eax
    mov [rbp - 8], r10d
    mov eax, [rbp + 16]
    mov edi, [rbp - 8]
    mov r10d, edi
    imul r10d, eax
    mov [rbp - 16], r10d
    mov eax, 1400
    mov edi, [rbp - 8]
    cmp edi, eax
    jle bb_1_leq_1
    mov r10d, 1
    jmp bb_1_afterGt_1
bb_1_leq_1:
    mov r10d, 0
bb_1_afterGt_1:
    cmp r10d, 0
    je else_2
    jmp then_2
then_2:
    mov eax, 10
    mov edi, [rbp - 16]
    mov r10d, edi
    add r10d, eax
    mov [rbp - 24], r10d
    jmp finally_2
else_2:
    mov eax, 10
    mov edi, [rbp - 16]
    mov r10d, edi
    sub r10d, eax
    mov [rbp - 24], r10d
    jmp finally_2
finally_2:
    mov eax, [rbp - 24]
    mov eax, eax
    mov rsp, rbp
    pop rbp
    ret
```

Listing F.1: Compiled function `foo` (see section 4.1) before optimizations.

---

```
foo:
    push rbp
    mov rbp, rsp
    sub rsp, 16
bb_1:
    mov eax, [rbp + 16]
    mov edi, 1440
    mov r10d, edi
    imul r10d, eax
    mov [rbp - 8], r10d
    mov eax, 10
    mov edi, [rbp - 8]
    mov r10d, edi
    add r10d, eax
    mov [rbp - 16], r10d
    mov eax, r10d
    mov rsp, rbp
    pop rbp
    ret
```

Listing F.2: Compiled function `foo` (see section 4.1) after optimizations.

## F. COMPILER OUTPUT EXAMPLES

---

```
bar:
    push rbp
    mov rbp, rsp
    sub rsp, 16
bb_1:
    mov eax, 42
    mov [rbp - 8], eax
    jmp guard_2
guard_2:
    mov eax, [rbp - 8]
    cmp eax, 0
    je finally_2
    jmp body_2
body_2:
    mov eax, [rbp + 16]
    cmp eax, 0
    je else_3
    jmp then_3
then_3:
    mov eax, 5
    mov [rbp - 16], eax
    jmp finally_3
else_3:
    mov eax, 10
    mov [rbp - 16], eax
    jmp finally_3
finally_3:
    mov eax, [rbp - 16]
    mov edi, 0
    mov r10d, eax
    push rdx
    mov edx, 0
    mov eax, edi
    idiv r10d
    pop rdx
    mov [rbp - 8], eax
    jmp guard_2
finally_2:
    mov eax, [rbp - 16]
    mov eax, eax
    mov rsp, rbp
    pop rbp
    ret
```

Listing F.3: Compiled function bar (see section 4.2) before optimizations.



---

```
bar:
    push rbp
    mov rbp, rsp
    sub rsp, 16
bb_1:
    mov eax, [rbp + 16]
    cmp eax, 0
    je else_2
then_2:
    mov eax, 5
    mov [rbp - 8], eax
    jmp finally_2
else_2:
    mov eax, 10
    mov [rbp - 8], eax
finally_2:
    mov eax, [rbp - 8]
    mov rsp, rbp
    pop rbp
    ret
```

Listing F.4: Compiled function bar (see section 4.2) after optimizations.

## F. COMPILER OUTPUT EXAMPLES

---

```
baz:
    push rbp
    mov rbp, rsp
    sub rsp, 16
bb_1:
    mov eax, 60
    mov edi, [rbp + 16]
    mov r10d, edi
    imul r10d, eax
    cmp r10d, 0
    je else_2
    jmp then_2
then_2:
    mov eax, 42
    mov [rbp - 8], eax
    jmp finally_2
else_2:
    mov eax, 10
    mov [rbp - 8], eax
    jmp finally_2
finally_2:
    mov eax, [rbp - 8]
    cmp eax, 0
    je else_3
    jmp then_3
then_3:
    mov eax, 1
    mov edi, [rbp - 8]
    mov r10d, edi
    add r10d, eax
    mov [rbp - 8], r10d
    jmp finally_3
else_3:
    mov eax, [rbp - 8]
    mov edi, [rbp + 16]
    mov r10d, edi
    add r10d, eax
    mov [rbp + 16], r10d
    jmp finally_3
finally_3:
    mov eax, [rbp - 8]
    mov edi, [rbp + 16]
    mov r10d, 60
    mov r14d, r10d
    imul r14d, edi
    mov edi, r14d
    sub edi, eax
    mov eax, edi
    mov rsp, rbp
    pop rbp
    ret
```

Listing F.5: Compiled function baz (see section 4.3) before optimizations.

---

```

baz:
    push rbp
    mov rbp, rsp
    sub rsp, 16
bb_1:
    mov eax, 60
    mov edi, [rbp + 16]
    mov r10d, edi
    imul r10d, eax
    mov [rbp - 16], r10d
    mov eax, r10d
    cmp eax, 0
    je else_2
then_2:
    mov eax, 42
    mov [rbp - 8], eax
    jmp finally_2
else_2:
    mov eax, 10
    mov [rbp - 8], eax
finally_2:
    mov eax, 1
    mov edi, [rbp - 8]
    mov r10d, edi
    add r10d, eax
    mov [rbp - 8], r10d
    mov eax, r10d
    mov edi, [rbp - 16]
    mov r10d, edi
    sub r10d, eax
    mov eax, r10d
    mov rsp, rbp
    pop rbp
    ret

```

Listing F.6: Compiled function baz (see section 4.3) after optimizations.