FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Assignment of master's thesis

Title:	Modular Compiler for the TinyC Language
Student:	Bc. Martin Prokopič
Supervisor:	Ing. Petr Máj
Study program:	Informatics
Branch / specialization:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2023/2024

Instructions

Familiarize yourself with the syntax and semantic of the TinyC language used in the NI-GEN course. Analyze the current C++ specific toolchain and determine how to extend it to support other implementation languages. As a proof of concept, design and implement optimizing compiler for the tinyC language in Scala compatible with the existing toolchain (including the tiny86 VM). Pay close attention to how advanced algorithms from compiler construction can be implemented in the Scala language in an idiomatic and clean way useful for educational purposes. Evaluate your design choices.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 24 November 2022 in Prague.

Master's thesis

Modular Compiler for the TinyC Language

Bc. Martin Prokopič

Department of Theoretical Computer Science
Supervisor: Ing. Petr Máj

May 4, 2023

Acknowledgements

First and foremost, I would like to thank my supervisor, Ing. Petr Máj, for his guidance and expertise in the field of compiler construction, which made this work possible. I would also like to extend my thanks to doc. Ing. Filip Křikava, Ph.D. for introducing me to the world of Scala and functional programming in general as part of the BIE-OOP and NIE-APR courses. Finally, I would like to thank my loving family, whose support has been instrumental in my academic and personal success.

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 accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 4, 2023 …………………

Czech Technical University in Prague
Faculty of Information Technology
© 2023 Martin Prokopič. 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

Prokopič, Martin. *Modular Compiler for the TinyC Language.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstrakt

Tato práce popisuje překladač jazyka tinyC do assemberu virtuálního stroje tiny86 (oboje používané v předmětu NIE-GEN). Překladač je implementován v jazyce Scala, je napojen na existující nástroje používané v předmětu, podporuje všechny konstrukty jazyka tinyC a pro generování kódu používá pokročilé algoritmy. Hlavní části překladače (frontend, middleend, backend) jsou jasně odděleny a lze je využívat buď ze Scaly nebo externě pomocí vlastního textového mezikódu ve stylu LLVM. Díky své jednoduchosti a modularitě je překladač vhodný pro výukové účely.

Klíčová slova překladač, pokrývání stromu, barvení grafu, tinyC, tiny86, Scala

Abstract

This thesis introduces an ahead-of-time compiler for the tinyC language targeting the tiny86 VM (both used in the NIE-GEN course) implemented in Scala. The compiler seamlessly integrates with the existing toolchain, supports all tinyC features and uses tree covering instruction selection and graph-coloring register allocation. The main modules (frontend, middleend, backend) are cleanly separated and can be extended either from Scala, or externally through a custom text-serializable LLVM-like SSA intermediate representation, which encourages use of the compiler for educational purposes.

Keywords compiler, tree covering, graph coloring, tinyC, tiny86, Scala

Contents

1 Introduction **1**
1.1 A Modular Compiler . . . 1
1.2 TinyC . . . 3
1.3 Tiny x86 . . . 4
1.4 Scala . . . 5
1.5 Previous Work . . . 6
1.6 Thesis Outline . . . 6

2 Intermediate-Code Generation and Optimization **7**
2.1 String Parsing Theory . . . 7
2.2 Lexical and Syntax Analysis . . . 15
2.3 Semantic and Type Analysis . . . 17
2.4 Intermediate Representation . . . 18
2.5 Compiling Source Code to IR . . . 22
2.6 Converting IR into SSA Form . . . 23
2.7 Optimizations . . . 24

3 Instruction Selection **29**
3.1 Overview of Instruction Selection Techniques . . . 30
3.2 Tree Covering . . . 30
3.3 Phi Node Elimination . . . 37
3.4 Macro Expansion . . . 39

4 Register Allocation and Assignment **43**
4.1 A Simple Local Register Allocator . . . 43
4.2 Liveness Analysis . . . 44
4.3 Register Allocation by Graph Coloring . . . 48
4.4 Linear Scan Register Allocation . . . 54
4.5 Post Processing . . . 54

5 Design and Implementation **57**
5.1 Design Goals . . . 57
5.2 Extending the NIE-GEN Toolchain . . . 58
5.3 Parsing and Typechecking TinyC . . . 60
5.4 Semantic and Type Analysis . . . 62
5.5 Intermediate Representation . . . 63
5.6 Compiling TinyC to IR . . . 68
5.7 The Optimizer . . . 70
5.8 Overview of the Backend . . . 71
5.9 Instruction Selection . . . 72
5.10 Register Allocation . . . 74
5.11 Post Processing . . . 76
5.12 Implementation . . . 77
5.13 Documentation . . . 81

6 Evaluation **83**
6.1 Unit Tests . . . 83
6.2 Integration and End to End Tests . . . 83
6.3 Benchmarks . . . 84

Conclusion **89**
Future Work . . . 89

Bibliography **91**

A Acronyms **97**

B User Guide **99**
B.1 Building . . . 99
B.2 Usage . . . 100
B.3 Supported TinyC Features . . . 102

C IR Grammar **103**

D Contents of the Electronic Attachment **107**

List of Figures

1.1 A modular compiler supporting 3 different languages and targets. . 2

2.1 δ_1 for grammar $G_1 = (\{E, T, F\}, \{+, *, (,), a\}, \delta_1, E)$. 8
2.2 δ_2 for grammar $G_2 = (\{E\}, \{+, *, (,), a\}, \delta_2, E)$. 8
2.3 A parse tree and two different derivations of $a + a$ in G_2. 9
2.4 Two possible parse trees for the sentence $a + a * a$ in G_2. 9
2.5 A modified grammar G_2' with removed left recursion and new start symbol S. 10
2.6 Productions of G_3 demonstrating the dangling-else problem. 12
2.7 A PEG $P_1 = (\{S, E, T, F, A\}, \{+, *, (,), a\}, R, S)$, where R is written above. 14
2.8 An example of (a) parse tree (as parsed by grammar from fig. 2.1) and (b) equivalent AST for expression $(a) + a * a$. 16
2.9 A simple C program demonstrating the limitations of tree-address code. (a) shows the original code and (b) the equivalent three-address code, where each statement does only a single operation. . 19
2.10 (a) the program from fig. 2.9b lowered into a set of basic blocks and (b) its basic block control-flow graph. `B1` is the entry point, `B4` is the only exit point. 20
2.11 A SSA form of the program from fig. 2.10a. We use the ϕ-node to join values of i coming from `B1` and `B3`. (b) then shows a DFG of the program, where the ϕ-node completes a cycle. 21
2.12 An example of a LLVM IR of a simple C program (a) before and (b) after optimizations (`clang-16 -O1`). Function inlining together with constant propagation have determined that `x` is always truthy, so the compiler could replace the `if` statement with its body. Strength reduction also replaced multiplication by two with a left shift. (c) shows the original C source code. 27

3.1 An intermediate representation of the expression $1 + 2 * 3$. 31

3.2 An example showing the effect of (b) edge splitting and (c) node duplication on a common subexpression. After edge splitting the UMul is evaluated only once and its result is stored in a register. . 36
3.3 An example of a shortcoming of naive ϕ-node elimination. (a) shows the original CFG, (b) shows the incorrect result of merging the x, y and z variables, which was corrected in (c) by inserting copy statements. 38
3.4 An example demonstrating the lost-copy problem when we hoist a copy across a critical edge. The program in (b) incorrectly prints the value of x from the last instead of the penultimate iteration. . 39
3.5 An example of the swap problem, the program in (b) is incorrect. . 40
3.6 A corrected version of (a) the lost copy and (b) swap problem examples by introducing temporary variables t_x and t_y. 40

4.1 Example CFG of a program for demonstrating liveness analysis. . 46
4.2 Hasse diagram of (a) a flat lattice and (b) a powerset lattice. . . . 47
4.3 A sample source code and a corresponding colored interference graph with three precolored physical registers r_1, r_2 and r_3. t_1 and t_2 do not interfere, because they are related only by a move. . 49
4.4 Diagram of phases of the graph coloring register allocation algorithm 49
4.5 Lifecycle diagram of a node during register allocation. 53

5.1 Two variants of integrating our compiler (tinycc.jar) with tiny86, in (a) tiny86 runs inside the JVM together with the compiler and in (b) the applications are separated on the OS level and communicate using files. 58
5.2 Compilation of a tinyC function which accepts and returns a `struct`. The structures are passed by reference and copied by the callee. . . 69
5.3 Diagram of the backend architecture. 71
5.4 Stack layout after entering a function. The first argument is at `[BP + 2]`, the first local variable is at `[BP + -1]`. 74
5.5 Spilling `VF7` requires a new integer temporary `VR9`, which the integer register allocator later assigns to `R0`. 74
5.6 Constructing a register allocator from the allocation algorithm, register-specific information and additional spilling-related code. . 80

List of Tables

2.1 FIRST and FOLLOW sets for G'_2. 11
2.2 A parsing table for grammar G_2. 11
2.3 A subset of TIP (equivalent to µC) type inference rules required for typechecking listing 2.2. I stands for an integer literal, E for an expression and X for a program variable (functions are treated as values) [1]. 18

3.1 A simple pattern set for a tree-based IR and tiny86-like target language. The `LEA` instruction can be used to perform combined multiply-add operation in a single instruction. 31

4.1 Steps of the iterative solver for the CFG from fig. 4.1. 46

5.1 Mapping between PEG expressions and our combinator library. . . 61

6.1 Comparison of size and performance of generated code with and without enabled SSA construction. With enabled optimizations, the code is in both cases smaller and the program finishes execution faster. 87
6.2 Comparison of size and performance of generated code by our compiler (tinycc) with a different, older compiler (tinyc) with only rudimentary instruction selection and a simple local register allocator. tinycc produces smaller and faster code for both programs. The numbers in the second column are different from table 6.1 because here they include the required `printnum` implementation. 87

List of Listings

1.1 An example of printing a string to standard output in tinyC using the builtin `print`. 4
1.2 An example of Scala 2 code with an immutable list and pattern matching. 5
2.1 An example of variable shadowing in tinyC and C since the C99 revision. 17
2.2 A simple µC program which demonstrates type inference. The type of x and y is not known until after the call expression. . . 18
3.1 An example maximal munch implementation in Scala for a subset of patterns from table 3.1 32
3.2 An example of two patterns from a PCC machine description. The first pattern matches a `+=` operator if the result should be stored in the A register, the first operand is an integer argument stored in A register and the second operand is a `NAME` node. If the match succeeds, the compiler emits the assembly code (after expanding the `AL` and `AR` macros) and replaces the matched IR subtree with the left subtree (`RLEFT`). The second pattern is used for matching `+`, `-`, `|`, `&` and `^` arithmetic operators and uses the `OI` macro and `|` operator for more concise notation. [2] 33
5.1 The VM crashed as a result of loading from invalid memory address -1 after it fetched the `MOV` instruction on address 5 even though it was never going to be executed. 59
5.2 An example of an automatically generated parser error message caused by an incomplete expression. 61
5.3 A simple combinator parser for PEG from fig. 2.7. 61
5.4 The IR of the “Hello, world!” program from listing 1.1. 68
5.5 An example Scala implementation of a rewrite rule for the IAdd instruction. 73
5.6 Tiny86 listing of the inserted function prologue and epilogue. . 76

5.7 Example of how the single-purpose classes can be combined to compile a tinyC source code into IR. We also provide a convenience function to do these steps at once. 78
5.8 Using Scala class composition to construct and run a tiny86 maximal munch instruction selector with the default ruleset. . . 80
6.1 factorize.c: A program for computing prime factors of a positive integer. 84
6.2 collatz.c: A program that prints the number of reduction steps required to reach 1 from a positive starting integer *n*. 85
6.3 A complete listing of `void collatz(int)` from collatz.c compiled by tinycc with optimizations. The tiny86 assembly does not support symbolic labels, so they are represented as comments. 88

Chapter 1

Introduction

The Compiler Construction (NIE-GEN) course teaches tens of FIT CTU students every year how to design and implement a compiler from start to finish, focusing mostly on the optimizer and target code generation (backend) parts. Each student is provided with a copy of the NIE-GEN Toolchain, which contains a parser for tinyC, a low-level C-like programming language [3], and the tiny86 virtual machine to be used as the target.

Although the tinyC parser is complete and the tiny86 VM has features that make it excellent for teaching purposes, the only way to interact with them is through a C++ API [4]. It means that without creating bindings for another language (which is a lot of additional work for an already extensive project) the student has to implement the entire compiler in C++.

The goal of this thesis is to design a modular compiler for tinyC in Scala (a modern high-level language [5]) that integrates well with the existing toolchain and which can be stripped down to be used as a foundation for NIE-GEN students to build their compiler projects on. We evaluate suitability of Scala by selecting and implementing some advanced techniques from compiler construction, chosen with the goal of exploiting most of the features provided by the tiny86 VM.

1.1 A Modular Compiler

Designing a compiler is a substantial project, which requires knowledge from multiple fields of computer science. To make the development process more tractable, we usually split the compiler into three parts – the frontend, the middleend (optimizer), and the backend. They work together to translate a source code in one language into another accepted by the target.

Programming in the (usually higher-level) source language is more pleasant and a good optimizing compiler can make better decisions than a programmer ever could if they have chosen to code in the target language directly.

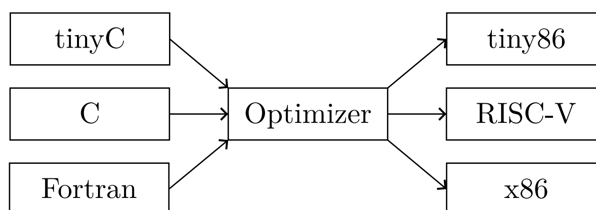


Figure 1.1: A modular compiler supporting 3 different languages and targets.

1.1.1 Frontend

The frontend is responsible for analyzing the source code and translating it into an intermediate representation that is then processed by the rest of the compiler. The frontend is the only part dependent on the source language.

Lexical Analysis First, a lexical analyzer scans the source code into a sequence of tokens (e.g. an identifier, an operator, or an integer or string literal) while also removing whitespace and comments.

Syntax Analysis A parser then uses a grammar to build an abstract syntax tree, which captures the structure of the program. This is a well-researched part of computer science and in section 2.1 we describe two techniques – LL(1) parsing and parser expression grammars.

Semantic Analysis The grammar used by the parser cannot capture all of the intricacies of the source language. The semantic analyzer links identifiers to their declarations, determines the type of each expression according to the rules of the language and optionally performs some additional checks.

Intermediate Code Generation The remainder of the frontend then takes the AST annotated with type and other semantic information and translates it into an unified intermediate representation (more in section 2.4). The compiler uses the IR as an abstraction over all supported source and target languages, which is the main tool enabling its modularity. Instead of developing $N \cdot M$ specialized compilers for each combination of N source and M target languages, we implement only N frontends and M backends that use a shared optimizer (see fig. 1.1).

1.1.2 Middleend (Optimizer)

The middleend takes IR produced by the frontend and optimizes it using language-independent passes. Thanks to the shared optimizer, both frontends and backends can be much simpler and easier to maintain. We mention optimizations such as SSA construction and constant propagation in sections 2.6

and 2.7, both of which can significantly improve efficiency of the generated code.

1.1.3 Backend

The last part of a compiler is the backend, which is responsible for generating the target code from the optimized intermediate representation produced by the previous stages.

Instruction Selection First, the instruction selector implements the IR program using the best combination of machine instructions in respect to execution speed, code size or some other criterion. We describe two popular techniques, macro expansion and tree covering, in chapter 3.

Register Allocation and Assignment The code generated by the instruction selector uses an unlimited amount of temporaries, but the target provides only a fixed amount of physical registers. The register allocator determines whether a temporary should be stored in a register or spilled to the (slower) memory. This is a NP-complete problem [6], but we show two heuristic methods producing good results in chapter 4.

Target-Dependent Optimizations After register allocation, the compiler has the last opportunity to perform target-specific optimizations. This is often a peepholer pass, which can make up for a simpler instruction selector by replacing common patterns in the target code with their optimal alternatives.

1.2 TinyC

The first part of the NIE-GEN toolchain is the tinyC parser. TinyC is a simplified subset of the C programming language, designed for teaching compilers and code generation. Just like C, it is a general-purpose programming language suitable for writing portable code, which can then be executed on different targets. It is a relatively “low level” language, which means that it does not deal with objects or more complex data structures and it leaves memory management up to the programmer.

To make it better suitable for teaching, it has been stripped as much as possible while keeping some parts interesting for compiler construction. It supports functions, the usual control flow statements (`if`, `for`, `while`, `switch`), local and global variables, pointers[1], structures and 1D arrays. However, the syntax has been simplified and only 3 basic data types have been included (`char`, `int` and `double`). We show an example program in listing 1.1. [3]

[1] including pointers to functions

The tinyC language reference specifies only the syntax, which allows students to express some creativity during the implementation. We design our compiler to adhere as much as possible to semantics of the regular C language as specified by the C99 standard [7], which makes our variant of tinyC more predictable to unfamiliar readers and enables easy translation between regular C and tinyC code.

```
int main() {
  char *str = "Hello,␣world!\n";
  while(*str) {
    print(*(str++));
  }
  return 0;
}
```

Listing 1.1: An example of printing a string to standard output in tinyC using the builtin `print`.

1.3 Tiny x86

Our tinyC compiler targets the tiny86 VM, designed by Ivo Strejc [4] for purposes of NIE-GEN. It simulates a CPU with a custom register-based ISA. There are two sets of registers, 64-bit-wide integer and double precision floating point registers, whose amount is configurable by the user. This configurability allows testing a compiler even before we implement a working register allocator. Tiny86 instructions can also access a data memory addressed in 64-bit words (Harvard architecture) and communicate with the user through standard input and output.

Targeting tiny86's simplified ISA allows the student to focus more on the compiler itself instead of studying the intricacies of a fully-fledged architectures like x86 or RISC-V. Still, many tiny86 instructions support multiple complex addressing modes, which puts it into the CISC territory.

Tiny86 is distributed as a shared C++ library, which exposes a simple builder interface to directly construct a program using instruction objects. It contains no built-in text- or binary-based assembly format, which makes integrating with non-C++ applications harder.

Luckily, Filip Gregor extended [8] the tiny86 interpreter with a proof-of-concept parser for a text-based program representation as part of his work on an interactive debugger. In chapter 5 we describe how we have used this parser to interface tiny86 with our tinyC compiler.

During execution, the VM collects information about lifecycle of each instruction as it goes through the CPU pipeline. Such statistics include the

number of memory accesses or the count of pipeline stalls[2], which allow us to measure quality of the code emitted by our compiler.

1.4 Scala

Scala is a modern high-level language, which combines object-oriented and functional programming concepts together in a statically typed language. The name Scala stands for scalable language and it was chosen because it grows with demands of the programmer. Novice users can quickly start developing programs and senior developers can fully utilize the advanced syntax constructs and extensibility offered by Scala. [5]

From functional language perspective, it offers a powerful type system, large immutable collection library and constructs designed to allow building code from smaller blocks. The object-oriented aspect helps with structuring large projects and defining clear abstractions and interfaces between components. Scala mainly targets the JVM and can seamlessly interoperate with other projects within the Java ecosystem[3]. [5]

Currently there are two simultaneously supported versions of Scala, numbered 2 and 3, where Scala 3 is fairly new (released in 2021) and a major overhaul of the previous version. Scala 2 is already taught on FIT CTU in the Object-Oriented Programming (BIE-OOP) and Static Program Analysis (NIE-APR) courses, so it is the natural choice given the educational aspect of this thesis. Listing 1.2 contains a small snippet of Scala 2 code demonstrating its functional and object-oriented nature.

```
sealed trait Animal
case object Cat extends Animal
case class Dog(name: String) extends Animal

List(Cat, Dog("Spot"), Dog("Buck")).foreach({
  case Cat => println("a␣cat:␣Meow!")
  case Dog(name) => println(s"$name:␣Woof!")
})
```

Listing 1.2: An example of Scala 2 code with an immutable list and pattern matching.

[2] A pipelined CPU stalls when the next instruction depends on completion of another in a previous stage that has not yet finished.

[3] There is also Scala.js, which allows Scala to run in the web browser and Scala Native, which is an ahead of time compiler to native executables.

1.5 Previous Work

The most popular compilers used today are GNU Compiler Collection (GCC) and Low Level Virtual Machine (LLVM), both supporting many languages and targets. They are comparable in terms of quality of the generated code, but GCC is built as a monolithic program with a fixed set of built-in supported languages and extending it means directly modifying its source code, which can be difficult to understand.

On the other hand, Lattner et. al designed LLVM [9] from the ground up with a common low-level intermediate representation and a framework for lifelong analysis and transformation of programs, which can be used by other software. If we wanted to implement a tinyC compiler using LLVM, we could just translate the source into a textual representation of the LLVM IR and let the optimizer and backend turn it into the target code.

The ahead-of-time microC (µC) compiler [10] designed by Král is comparable in scope to our thesis. The µC language is similar in design and purpose to tinyC, but Král's thesis focuses on the optimizer part of a compiler, while in our thesis we explore interesting ideas in all parts of a modular compiler with emphasis on code generation in the backend.

1.6 Thesis Outline

In the following three chapters we describe techniques currently used in compiler construction (chapter 2 – frontend & optimizer, chapter 3 – instruction selection, and chapter 4 – register allocation).

In chapter 5 we then choose a subset of the described methods and explain how we have used them to design and implement our own modular tinyC compiler targeting the tiny86 VM.

In chapter 6 we evaluate correctness and performance of our implementation and in the last chapter we conclude our thesis and suggest possible future improvements.

Chapter 2

Intermediate-Code Generation and Optimization

In this chapter we will describe the first two parts of a typical modular compiler, which were introduced in section 1.1 – the frontend and optimizer.

Before the compiler can translate the source program into the target language, it must first understand its structure (syntax) and meaning (semantics). To make the syntax analysis easier to implement and reason about, we do not use characters of the source code directly. Instead, we first processes them into a stream of tokens using lexical analysis. Before we can design a simple lexical and syntax analyzer, we first have to define some basic principles of the string and parsing theory. A good resource with more complete definitions and explanations is the book by Aho et al. [6], known as "the Dragon Book", which we have also used while writing this chapter.

2.1 String Parsing Theory

Definition 1. A *string* is a finite sequence of symbols from a finite set called the *alphabet* (usually denoted Σ). The symbol ε means an empty string.

2.1.1 Context-Free Grammar

Definition 2. A context-free grammar is a 4-tuple (N, Σ, δ, S), where

- N is a finite set of nonterminals (syntactic variables),
- $\Sigma, \Sigma \cap N = \emptyset$ is a finite set of terminals called the alphabet,
- $\delta \subseteq N \times (N \cup \Sigma)^*$ is a finite set of rewrite rules (*productions*), where each has a single nonterminal on the left side and a (possibly empty) string of nonterminals and terminals on the right, and

- $S \in N$ is the start symbol.

A grammar is a precise specification of syntax of a given language. It is structured, yet easy-to-understand and there exist tools to automatically construct a syntax analyzer from some classes of grammars. That can be advantageous when we want to extend the grammar later in development of the compiler. In fig. 2.1 we show a set of productions for a grammar $G_1 = (\{E, T, F\}, \{+, *, (,), a\}, \delta_1, E)$, which describes a simple language for arithmetic expressions.

For ease of understanding, we will use uppercase characters for nonterminals and we may use | to join alternative productions for a single nonterminal.

$$E \to E \ + \ T \mid T$$
$$T \to T \ * \ F \mid F$$
$$F \to (\ E \ ) \mid a$$

Figure 2.1: δ_1 for grammar $G_1 = (\{E, T, F\}, \{+, *, (,), a\}, \delta_1, E)$.

By applying a sequence of productions of a grammar as rewrite rules beginning with the start symbol, we construct a derivation. In each step, we replace a single nonterminal using one of the productions of the grammar. We will now use a simplified version (fig. 2.2) of the previous grammar. By applying the rule $E \to (\ E \ )$ we get $E \Rightarrow (\ E \ )$ and say that E *derives* $(\ E \ )$. We say that $(\ E \ )$ is a *sentential form*, because it contains the nonterminal E, which can be derived further. We can for example continue with the rule $E \to a$ to get $E \Rightarrow (\ E \ ) \Rightarrow (\ a \ )$. There are now no more nonterminals left and the derivation has proven that $(\ a \ )$ is one of the strings in the language produced by the grammar G_2.

2.1.2 Parse Tree

A *parse tree* is another way to prove that a sentence (a string of terminals) can be derived by a grammar, it however does not show the exact order of derivations. The root and interior nodes of the parse tree are labeled with nonterminals representing a left side of a production and their children are nodes labeled left-to-right with symbols of the corresponding right side. The

$$E \to E \ + \ E \mid E \ * \ E \mid (\ E \ ) \mid a$$

Figure 2.2: δ_2 for grammar $G_2 = (\{E\}, \{+, *, (,), a\}, \delta_2, E)$.

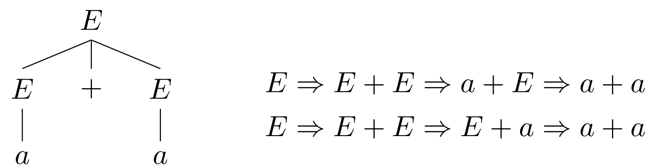
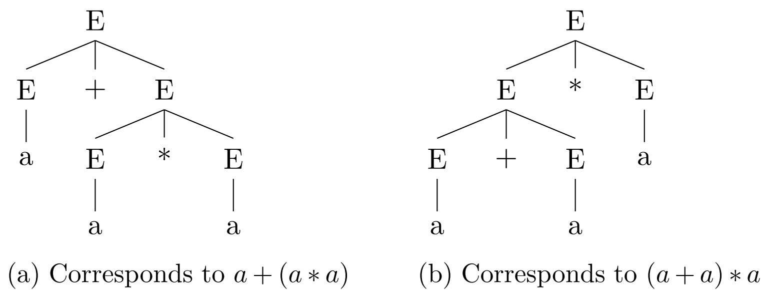


Figure 2.3: A parse tree and two different derivations of $a + a$ in G_2.



(a) Corresponds to $a + (a * a)$ (b) Corresponds to $(a + a) * a$

Figure 2.4: Two possible parse trees for the sentence $a + a * a$ in G_2.

leaves of the tree contain terminals and nonterminals, which when read left-to-right denote a sentential form or a sentence.

For a given sequence of derivations, we can construct an unique parse tree and from a parse tree we can construct (possibly multiple) sequences of derivations [6]. We can choose to produce leftmost or rightmost derivations to make this relationship one-to-one – we always expand either the leftmost or rightmost nonterminal in a sentential form. In fig. 2.3 we show an example parse tree and two possible sequences of derivations for the same sentence.

2.1.3 Ambiguity of a Grammar

The last term we need to define before we can proceed with description of parsing techniques is ambiguity of a grammar. The grammars G_1 (fig. 2.1) and G_2 (fig. 2.2) describe the same language, but the second one is ambiguous. A grammar is *ambiguous* if it produces more than one parse tree for some sentence s. In another words, there are multiple possible leftmost and rightmost derivations of s. We use the nonterminals E, F and T in G_1 to capture the priority of $+$ and $*$, whereas in G_2 there is only a single nonterminal E. In fig. 2.4 we show two different parse trees of the sentence $a + a * a$ in G_2.

Compilers usually use grammars that are either unambiguous, or the parsers contain additional disambiguating rules that always select one of the possible parse trees for further processing.

2.1.4 Top-Down Parsing

Top-down parsing is one of the two most common approaches to constructing a parse tree for a given sentence, the other one being bottom-up parsing. A top-down parser tries to find a *left parse* (a sequence of leftmost derivations) by building the parse tree from the root and continuing in preorder (depth-first) fashion. This is often implemented as a *recursive-descent* parser.

At each step the parser needs to select a production to use for the current (leftmost) nonterminal. In the most general way it can use backtracking to try all possibilities, but that is not used in practice because of exponential time complexity.

$LL(k)$ parsers make this decision by looking ahead at the next k symbols in the unconsumed input, for that reason they are called *predictive parsers*. Whenever a $LL(k)$ parser makes a decision, it never backtracks. That greatly limits the class of grammars it can be constructed for (and therefore languages it can accept), but it is still usually generic enough for programming languages.

A left-recursive grammar like the one in fig. 2.1 cannot be used to construct a recursive-descent parser (even one with backtracking), because it would get stuck in an infinite loop while expanding the nonterminal E or T. Luckily, left recursion can be removed by introducing new nonterminals [6] like in fig. 2.5. We have also added a new start symbol S, which matches the original E terminated by an end-of-input marker \$. This will be required later during parsing table construction.

$$
\begin{aligned}
S &\to E \ \$ \\
E &\to T \ E' \\
E' &\to + \ T \ E' \mid \varepsilon \\
T &\to F \ T' \\
T' &\to * \ F \ T' \mid \varepsilon \\
F &\to (\ E \ ) \mid a
\end{aligned}
$$

Figure 2.5: A modified grammar G'_2 with removed left recursion and new start symbol S.

To construct a predictive parser from a grammar, we need to define two functions, FIRST(α) and FOLLOW(A).

Definition 3. FIRST : $(N \cup \Sigma)^* \to \Sigma \cup \{\varepsilon\}$ returns the set of all possible terminals that may appear at the beginning of sentences derived from the sentential form α. If an empty string can be derived from α in some number of steps, ε is also included.

Definition 4. FOLLOW : $N \to \Sigma \cup \{\varepsilon\}$ returns the set of terminals that may appear immediately after the nonterminal A in some sentential form.

We can easily compute both FIRST and FOLLOW from the grammar definition [6] and use them to build a predictive parser. We have included a sample of values for G'_2 in table 2.1.

Rule ($A \to \alpha$)	FIRST(α)	FIRST(A)	FOLLOW(A)
$S \to E\ \$$	$\{(, a\}$	$\{(, a\}$	$\{\varepsilon\}$
$E \to T\ E'$	$\{(, a\}$	$\{(, a\}$	$\{), \$\}$
$E' \to +\ T\ E'$	$\{+\}$	$\{+, \varepsilon\}$	$\{), \$\}$
$E' \to \varepsilon$	$\{\varepsilon\}$		
$T \to F\ T'$	$\{(, a\}$	$\{(, a\}$	$\{+,), \$\}$
$T' \to *\ F\ T'$	$\{*\}$	$\{*, \varepsilon\}$	$\{+,), \$\}$
$T' \to \varepsilon$	$\{\varepsilon\}$		
$F \to (\ E\ )$	$\{(\}$	$\{(, a\}$	$\{*, +,), \$\}$
$F \to a$	$\{a\}$		

Table 2.1: FIRST and FOLLOW sets for G'_2.

The grammar G'_2 is $LL(1)$ – only a single symbol look-ahead is required to choose the correct production. Given the grammar and the values of FIRST and FOLLOW functions, we can now compute the *parsing table* using the algorithm from [6]. The result for G_2 is shown in table 2.2. Rows of the table correspond to nonterminals, the columns contain the next symbol in the input and the cells hold the production rule that the parser should select. The parsing table can then be easily translated to implementation code (a procedure for each nonterminal containing a `switch` statement for look-ahead).

	a	$+$	$*$	$($	$)$	$\$$
S	$S \to E\ \$$			$S \to E\ \$$		
E	$E \to T\ E'$			$E \to T\ E'$		
E'		$E' \to +\ T\ E'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to F\ T'$			$T \to F\ T'$		
T'		$T' \to \varepsilon$	$T' \to *\ F\ T'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to a$			$F \to (\ E\ )$		

Table 2.2: A parsing table for grammar G_2.

If the parsing table contains an unique rule in each cell, the grammar is $LL(1)$. Otherwise, we would either have to perform some more reductions of the grammar [6] or use a different parser family altogether.

An ambiguous grammar can never be $LL(k)$, but sometimes the best solution is to manually pick one option in the parsing table. Many programming languages with the `if` statement deal with a so-called *dangling else* problem. See an example grammar $G_3 = (\{S', S, E\}, \{\textbf{if}, \textbf{then}, \textbf{else}, \mathit{expr}, \mathit{stmt}, \$\}, \delta_3, S')$ with δ_3 in fig. 2.6.

$$
\begin{aligned}
S' &\to S\$ \\
S &\to \textbf{if}\ \mathit{expr}\ \textbf{then}\ S\ E \\
S &\to \mathit{stmt} \\
E &\to \textbf{else}\ S \mid \varepsilon
\end{aligned}
$$

Figure 2.6: Productions of G_3 demonstrating the dangling-else problem.

If we tried to construct a parsing table for G_3, it would contain duplicate entries for E and **else**. The grammar does not specify if the **else** statement should belong to the closest **if** (usually the desired behavior) or not, see for example the sentence **if** *expr* **then if** *expr* **then** *stmt* **else** *stmt*. We can fix this without changing the grammar by manually modifying the parsing table to prefer $E \to \textbf{else}\ S$.

2.1.5 Parser Expression Grammar

A context-free grammar describes how sentences of the language can be produced. But when developing a parser, we need to recognize how an already existing sentence was derived. This clash of directions led Ford [11] to design a formalism for parser expression grammars (PEGs), which are used to more directly describe a top-down parser.

Definition 5. A PEG is a 4-tuple (N, Σ, R, e_S) where

- N is a finite set of nonterminals,
- $\Sigma, \Sigma \cap N = \emptyset$ is a finite set of terminals,
- R is a total function from nonterminals to *parsing expressions*, and
- e_S is the initial parsing expression.

We write the rules given by R as $A \leftarrow e$.

R being a total function means that for each nonterminal $A \in N$ there is exactly one parsing expression $R(A)$. A parsing expression receives an input string of terminals and either fails, or succeeds and consumes a prefix of the

input, leaving a remaining suffix. We define parsing expressions inductively, the syntax is as follows:

$$e = \varepsilon \mid a \mid A \mid e_1e_2 \mid e_1/e_2 \mid e^* \mid !e$$

- ε matches the empty string (and does not consume any input),
- $a, a \in \Sigma$ matches and consumes the terminal a,
- $A, A \in N$ tries to match $R(A)$,
- e_1e_2 (a sequence) matches e_1 and tries to match e_2 on the remaining input,
- e_1/e_2 (prioritized choice) tries to match e_2 and only if it fails tries to match e_2,
- e^* greedily[4] matches zero or more repetitions of e,
- $!e$ (a not-predicate) fails if e succeeds, otherwise if e fails $!e$ succeeds without consuming any input.

In comparison with $LL(k)$ languages, PEGs have more expressive power. A PEG can express all $LL(k)$ and deterministic $LR(k)$ languages and even some non-context-free languages, thanks to the limited backtracking ability and the not-predicate [11]. Despite this, they can be parsed in linear time with the help of memoization [12].

We have rewritten the simple expression grammar from section 2.1.4 as a PEG in fig. 2.7. Like with $LL(1)$ parsing, we need to explicitly model end of input, because parsing expressions match on a prefix of the input. Instead of appending a terminator the input string, we use the expression $!A$, where A matches any terminal. Parsing expressions also cannot contain left-recursion, but we can perform the same transformation as with $LL(1)$ grammars, or use the repetition expression (e^*).

A natural way to implement recursive descent parsers in functional languages (including Scala [5]) is using higher-order functions called parser combinators[5]. A combinator parser as designed by Hutton [13] is a function from an input string of symbols to a list of result values and remaining suffixes. That is, it performs full backtracking and returns a list of all possible matches. If we modify the parser to return a `Maybe` (`Option` in Scala) instead of a list, we get the same semantics as a PEG [14]. The Scala standard parser combinator library does exactly that and includes some additional combinators to limit backtracking to improve performance [5].

[4]as many times as possible

[5]A parser *combinator* is a higher-order function, which takes one or more parsers as arguments and combines them into a new parser.

$$
\begin{aligned}
S &\leftarrow E\ !A \\
E &\leftarrow T\ (+\ T)^* \\
T &\leftarrow F\ (*\ F)^* \\
F &\leftarrow (\ E\ )\ /\ a \\
A &\leftarrow \textit{any terminal}
\end{aligned}
$$

Figure 2.7: A PEG $P_1 = (\{S, E, T, F, A\}, \{+, *, (,), a\}, R, S)$, where R is written above.

We build a combinator parser from smaller parsers using combinators, which perform the same basic operations as the operators in a parser expression (a sequence, prioritized choice, repetition and not-predicate). [13]

2.1.6 Error Reporting

When a parser is unable to recognize some input, we would usually want it to report the probable location of the syntax error. In some cases it could be beneficial if it tried to recover and continue parsing anyway, so we can display all found errors to the user at once.

In this section we will focus on error reporting, because error recovery is more complicated and can result in cascading errors [14].

Error reporting for LL parser (and LR for that matter) is simple, because it processes the input from start to end without backtracking. If the parsing table does not specify any transition for the next terminal, the parser aborts with an error at the current location in the input [6]. However, when a PEG or combinator parser reports a failure, the original source and location of the error is usually lost, because the parser could have backtracked and tried other alternatives [15].

Ford [15] implemented a simple heuristic, which simulates the error reporting of predictive parsers by passing information about all errors that occurred at the furthest point of the input string up the chain of parser expressions. The final error message contains information about the location, next symbols in the input and all expected symbols, but it is missing any additional context.

Maidl [14] proposed an extension of the PEG grammar with *labeled failures*, which work similarly to exceptions in programming languages. He added a new expression $\Uparrow^l$, which always fails with the label l and extended the ordered choice operator $e_1/^S e_2$ to backtrack only if e_1 fails with a label $l \in S$. The primitives emit the `fail` label on failure by default, but every parsing expression can be annotated to convert those `fail` labels to a custom label using the new `try..catch`-like operators. Together, this gives the author of

the grammar more control over backtracking and the ability to supply custom context-rich error messages.

Scheidecker [16] mentioned in his blog post a simple improvement of Ford's technique. Instead of keeping the expected symbols in a flat structure, he uses an expectation tree that is extended with friendly description l using new describeParser(e, l) combinator. When the inner parser e fails without matching any input[6], it overwrites its expectation with l. If e failed after matching some input, it wraps the expectation of e with a new node labeled with l.

Once the failure is propagated to the root and becomes fatal, the parser has the expectation tree for the furthest visited point of the input. Scheidecker uses the description of each leaf to list the expected symbols/tokens and the description of the innermost ancestor of all leaves as the context to build the error message.

2.1.7 Other Parsing Methods

In addition to $LL(k)$ and PEG parsing, we will use this section to briefly mention a few other methods.

Bottom-up parsing constructs a parse tree starting from the leaves and continuing up until the root. The $LR(k)$ class of grammars can be parsed by a shift-reduce parser, which is the most commonly used bottom-up parser style. They are table-driven like LL parsers and can be implemented efficiently while recognizing most context-free grammars for programming languages (the $LR(k)$ class of grammars is a proper superset of $LL(k)$). One disadvantage of LR parsers is that they are too complex to be built by hand without help of automatic tools. [6]

Algorithms exist for testing whether a string can be generated by a generic context-free grammar, but they have high time complexity, which makes them unsuitable for use in compiler construction [6].

2.2 Lexical and Syntax Analysis

The first thing a compiler frontend does is scanning and grouping of the input characters into *lexemes* and production of stream of meaningful tokens. A token has a name and an optional attribute and it is used as a terminal symbol during syntax analysis. A lexer of a typical C-like language recognizes the following classes of tokens [6]:

- tokens for every program keyword (such as `if`, `while`...),
- tokens for operators and other special symbols (`+`, `==`, `->`, `[`, `)`...),

[6] Remember that we work with the furthest visited location.

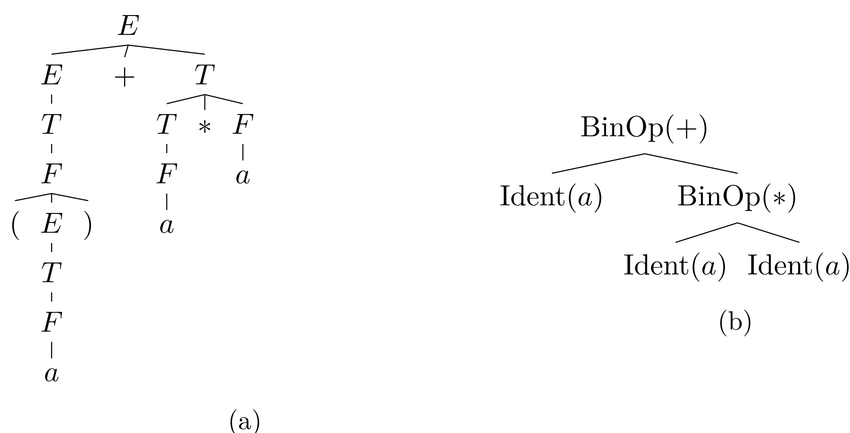


Figure 2.8: An example of (a) parse tree (as parsed by grammar from fig. 2.1) and (b) equivalent AST for expression $(a) + a * a$.

- a token representing an identifier – the name is stored in the attribute,
- a token representing a constant (an integer, a real number...),
- a token for string literal (`"foo"`, `'\n'`...).

We implement a lexer greedily, so it always matches the longest possible tokens, because otherwise it could incorrectly recognize identifiers such as `iface` as keywords (`if`). For that reason, CFGs are not suitable for describing the patterns of tokens. Instead, we can use ad-hoc algorithms, PEG parsers or their subset called regular expressions in a loop to parse a prefix of the input into a single token.

The syntax analyzer is then a top-down or bottom-up parser based on the theory described in section 2.1. During parsing, we call actions to construct an abstract syntax tree (AST). An AST is a data structure similar to a parse tree, but it hides the nuances of the grammar and keep only the syntax information useful for further processing.

The interior nodes of a parse tree represent nonterminals, whereas in an AST, they represent programming constructs. For the expression grammar from fig. 2.5, we would use only two node types instead of 6 nonterminals – a binary operator $(+, *)$ and an identifier (a). We compare a typical parse and syntax trees for $(a) + a * a$ in fig. 2.8.

An alternative to separate lexical and syntax analysis is to build an unified PEG parser and grammar. This is possible thanks to the greedy nature of PEG [11], but leads to a more complex grammar and a slower parser. The advantage of this approach is that we can easily express more complex lexing rules (for example the » at the end of a nested C++ template[7] [11]) and we

[7] Older versions of C++ mandated the space in `vector<vector<int> >`, so it is not confused with the `>>` bitwise shift operator.

can even embed one PEG parser into another (for example to parse JavaScript embedded in a HTML document [14]).

2.3 Semantic and Type Analysis

The parser hands off the AST to the semantic analyzer, which links identifiers to their declarations and checks type of each expression. It then augments the AST with the collected information and passes it to the intermediate code generator.

The semantic analyzer recursively visits every AST node in order and tracks the identifiers accessible from the current scope in *symbol tables* (*environments*). Whenever an identifier enters the current scope through a declaration, we add the symbol and a reference to the declaration (*a binding*) into the active environment. We then discard the binding when the declaration leaves the current scope. Whenever we visit an identifier, we look up its declaration in the active environment or return an error if there was no match.

Some languages, such as tinyC [3] or C since the C99 revision [7] have lexical scoping, which allows *shadowing* of multiple declarations of the same identifier (see listing 2.1). In case of lexical scoping, we store the environments in a stack, otherwise one or two shared symbols tables (one for global and one for local scope) suffice.

```
int x = 1;
do {
  int x = 2; // shadows the previous declaration
  print(x); // prints 2
} while(0);
print(x); // prints 1
```

Listing 2.1: An example of variable shadowing in tinyC and C since the C99 revision.

2.3.1 Type Analysis

During type analysis, the compiler assigns a type to each program expression (represented by an AST node) and checks that the types conform to a set of logical rules (*type system*) dependent on the source language. We use type checking to reduce (or completely eliminate) the need for dynamic checking for type errors and to aid intermediate code generation. [6]

In most statically typed languages (including C [7] and tinyC), we define a type of an expression in terms of types of its subexpressions (*type synthesis*). We can check the rules in this case by recursively visiting the abstract syntax tree. Whenever we cannot determine a type of an expression, we raise a type error. A typical rule can look like this: “if the current node is `+`, the left child

has type `int` and the right child has type `double`, then the result has type `double`."

Other languages, such as ML or µC[8] [17] are strongly typed, however the types are automatically *inferred* by the compiler without relying on the user. The types also cannot be computed using a single bottom-up pass over the AST, see for example a small µC program in listing 2.2. The typechecker does not know the type of the arguments of the `sum` function until it is called. For this reason, we build a list of *type constraints* for the whole program and then solve them using *unification* [1].

```
select(c, x, y) { var r; if(c) r = x; else r = y;
  return r; }
main() {
  var a, b, r;
  a = 2; b = 3;
  return select(1, a, b);
}
```

Listing 2.2: A simple µC program which demonstrates type inference. The type of x and y is not known until after the call expression.

We define constraints for each expression in terms of type variables ($[\![E]\!]$ holds the type of E). We list a few rules in table 2.3. Description of the constraint solver itself is outside of the scope of this thesis and can be found in [1] or [6].

$$\begin{aligned} I\text{:} \quad & [\![I]\!] = \texttt{int} \\ X = E\text{:} \quad & [\![X]\!] = [\![E]\!] \\ \texttt{if}\ (E)\ S_1\ \texttt{else}\ S_2\text{:} \quad & [\![E]\!] = \texttt{int} \\ X(X_1, \ldots, X_n)\{\ldots \texttt{return}\ E;\}\text{:} \quad & [\![X]\!] = ([\![X_1]\!], \ldots, [\![X_n]\!]) \to [\![E]\!] \end{aligned}$$

Table 2.3: A subset of TIP (equivalent to µC) type inference rules required for typechecking listing 2.2. I stands for an integer literal, E for an expression and X for a program variable (functions are treated as values) [1].

2.4 Intermediate Representation

In a modular compiler, the translation between the source code and compilation target is not direct. Instead, the compiler uses one or more intermediate representations (IRs), which are easier to analyze and optimize and act as a middleman between the frontend and the backend. A typical intermediate representation emitted by the frontend resembles instructions of a low-level

[8] a programming language designed for the static program analysis (NI-APR) course based on the TIP language from Møller's book [1]

```
int i = 0;
while(i + 1 < 2) {
  print(i * 2);
  i += 1;
}
```

(a)

```
int i = 0;
while(t1 = i + 1, t1 < 2) {
  t2 = i * 2;
  print(t2);
  i = i + 1;
}
```

(b)

Figure 2.9: A simple C program demonstrating the limitations of tree-address code. (a) shows the original code and (b) the equivalent three-address code, where each statement does only a single operation.

abstract machine. We design the IR to be easily producible from all of the supported source languages (can express all of their features in some way) and so it can be easily translated into all supported target languages. [6]

We can classify the form of an IR as either flat or hierarchical (usually a tree or a graph).

2.4.1 Flat IR

The flat variant has the form of a list of instructions, which are executed sequentially and the control flow is specified by labels and branch instructions. Each instruction performs only one operation (three-address code) and stores the result to a specified temporary variable, of which there is usually unlimited amount available. Figure 2.9 shows a simple program and its three-address code representation, which we will use as a running example. [18]

We group sequences of instructions, which always get executed together without any branches into *basic blocks*. A basic block ends with a terminator (usually an (un)conditional branch, return or a halt instruction), which specifies the next basic block to be executed.

To be able to reason better about the control flow of instructions or basic blocks in analyses and optimizations, we define the control-flow graph.

Definition 6. A control-flow graph (CFG) is a directed graph, where nodes correspond to instructions (or basic blocks) of a program. An edge (u, v) exists iff instruction (basic block) v can be executed directly after instruction (basic block) u. We mark one node as the *entry point* (entry(cfg)) of the program and a set of nodes as *exit points* (exit(cfg)). A node n has a set of predecessors (pred(n)) and successors (succ(n)) determined by its incoming and outgoing edges.

We would like to build a similar graph representation for the flow of values, but for that we need a form of IR where every variable has only one unique

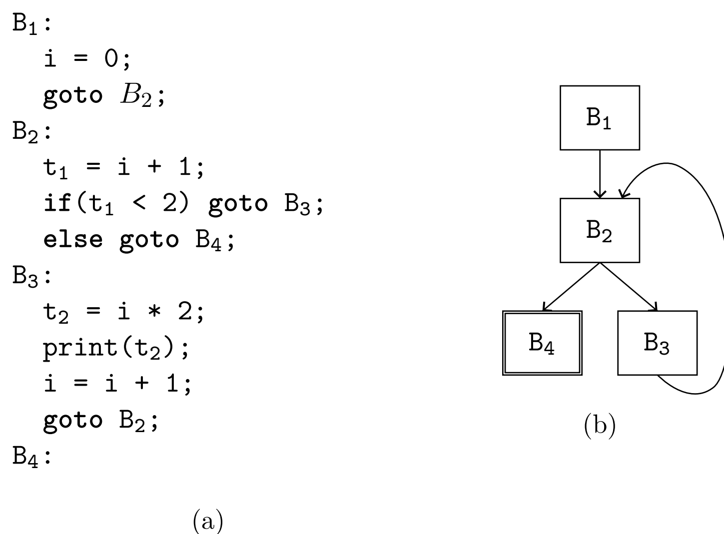


Figure 2.10: (a) the program from fig. 2.9b lowered into a set of basic blocks and (b) its basic block control-flow graph. $\texttt{B}_1$ is the entry point, $\texttt{B}_4$ is the only exit point.

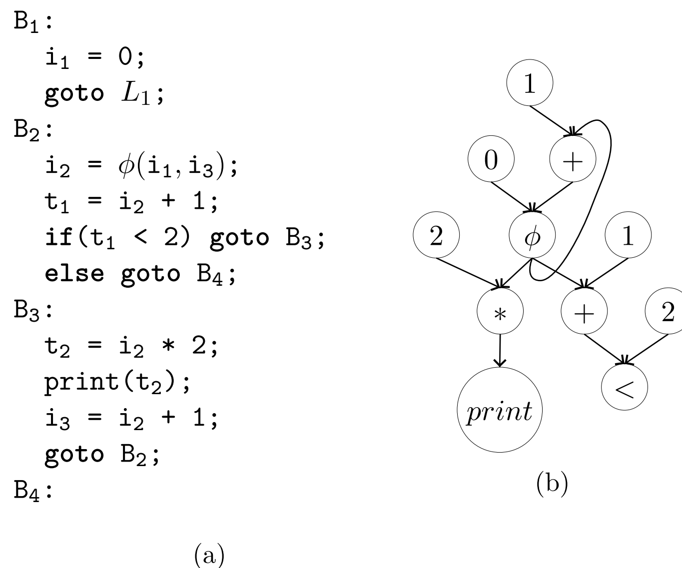
definition. We call this extension of three-address code the single static assignment (SSA) form. A distinctive aspect of SSA form is the presence of ϕ-nodes to combine two definitions of a variable coming from different predecessor blocks. This is a notational convention rather than a real instruction, which we later again eliminate during instruction selection or register allocation. See fig. 2.11 for an example of three-address code and its SSA form.

We can construct SSA form for every program by giving variables unique names and inserting ϕ-nodes in appropriate places. We will describe algorithms for that in section 2.6.

The single unique definition property of SSA allows us to visualize the data dependencies of instructions using a data-flow graph.

Definition 7. A data-flow graph (DFG) is a directed graph, where nodes correspond to instructions in a program and edges to their data dependencies. An edge (u, v) means that the instruction v uses (depends on) the result of instruction u.

In a valid DFG, every cycle must go through some ϕ-node. Otherwise none of the instructions in the cycle can have their requirements satisfied and begin executing. The data-flow graph is used by various dataflow analyzes, which we will mention in section 2.7.



```
B1:
  i1 = 0;
  goto L1;
B2:
  i2 = ϕ(i1,i3);
  t1 = i2 + 1;
  if(t1 < 2) goto B3;
  else goto B4;
B3:
  t2 = i2 * 2;
  print(t2);
  i3 = i2 + 1;
  goto B2;
B4:
```


Figure 2.11: A SSA form of the program from fig. 2.10a. We use the ϕ-node to join values of i coming from `B1` and `B3`. (b) then shows a DFG of the program, where the ϕ-node completes a cycle.

2.4.2 Tree-Based IR

Another common way to represent IR instructions is as tree nodes. A tree-based intermediate representation looks similarly to an abstract syntax tree, but the operations are much simpler (like in the flat three-address code from section 2.4.1). The hierarchy of a tree is closer to the typical programming language, but the tree representation of control flow is impractical for analyses and optimizations.

Appel [19] describes implementation of a simple optimizing compiler and uses trees as an intermediate representation. Instead of the high level control flow nodes like `if` and `while`, he defines a system of labels and jumps similar to a flat IR. In a later stage of the compilation, he splits the tree into smaller trees (one for each program statement) and groups sequences of the resulting trees into basic blocks.

2.4.3 Intermediate Representation in Existing Compilers

LLVM The core of the LLVM framework is built around a flat RISC-like IR. The instructions operate on SSA virtual registers and their control flow is given explicitly by listing them in basic blocks. The frontends do not have to emit SSA directly, instead they can use the available memory instructions (e.g. `alloca`, `load` and `store`), most of which the compiler then eliminates in a SSA construction pass (mem2reg). Figure 2.12 contains example of LLVM

IR code before and after optimization. [9]

GCC The main intermediate representation used in GCC are GENERIC and GIMPLE. The frontends target the GENERIC IR, which a tree representation with a structure similar to C source code. The middleend then lowers this IR into the GIMPLE representation, which is a standard three-address code. [20]

JVM The Java Virtual Machine uses a stack-based IR, which is a form somewhere between flat and hierarchical. Control flow is modeled using labels and jumps like in flat intermediate representation, but the operations do not use variables. Instead, they pop their operands from the implicit operand stack and push back their result. [18]

2.5 Compiling Source Code to IR

After we have determined the type of every program expression, emitting the IR code for a low level language like C is fairly straightforward and some compilers do it at the same time as type checking or parsing. A *single-pass* compiler (such as Bellard's TCC [21]) takes this one step further and skips the intermediate representation altogether and emits the target code directly. In this kind of a compiler, all three parts (frontend, optimizer, backend) run at once.

The IR represents the program statements at a lower level, so we refer to the compilation process as *lowering*. In this stage we usually lose some type information (opaque pointers) and accessing an array element or a `struct` member may require computing its offset if the IR does not support complex types directly.

We compile an expression either as a *l-value* ("location") or a *r-value* ("value") depending on what side of an assignment it lies at. The compiler has to statically check this, because not all expressions have a location and can be assigned to (for example `42`, `&foo` or `foo * bar` are all strictly r-values in C).

If the IR is strictly SSA, it cannot represent local variables directly and we need to perform online SSA construction, for which we describe a method in section 2.6.

When we compile control flow statements (`if`, `while`...), we need to reference a basic block which we have not reached yet and thus do not know its address. We can solve this by building the IR in memory and *back-patching* the jump statements after we know the location of all required blocks.

2.6 Converting IR into SSA Form

Single static assignment construction is one of the more important transformations that compilers often implement. Having the IR in SSA form makes analyses more efficient and easier to implement, because each variable has only a single definition.

A challenge arises if a basic block has multiple predecessors and more than one definition of a variable can reach a given program point. In that case we need to insert ϕ-nodes into the joining points. We will now introduce two methods that try to insert as few ϕ-nodes as possible.

We can perform SSA conversion either *offline* after we have the complete CFG available, or *online* during the IR code emission. The first algorithm for efficient SSA construction was designed by Cytron et al. [22] and uses the offline approach. The algorithm is rather complex and relies on several other analyses and transformations (liveness analysis, computation of dominance trees and frontiers), but it guarantees *minimality* of the SSA form in terms of number of ϕ-nodes. Because of its complexity, we will not explain it any further in this thesis and instead provide a simpler but equally good solution. We refer the reader either to the original paper, or to [6, 19] which also include (maybe more tractable) description of the algorithm.

Braun et. al. [23] proposed a simple online-first (but adaptable for offline use, see section 5.7.1) algorithm for SSA construction and its combination with on-the-fly optimizations. The algorithm works backwards by tending a map of variable definitions for each basic block.

2.6.1 Local Value Numbering

During IR code emission (in program execution order), we pay a special attention to instructions that *define* (write) or *use* (read) a variable. When we encounter an assignment into a variable v, we update the current definition of v in the current basic block in the map with reference to the right side of the assignment (WriteVariable in alg. 1). When the variable is later read, we use the map to look up its definition (ReadVariable). We call this process *local value numbering* [19]. After we process all instructions in a basic block, we mark the block as *filled*. An issue occurs when a variable is read for which we do not have any definition in the current basic block.

2.6.2 Global Value Numbering

We solve read of variable undefined in the current basic block by recursively looking for its definition in the predecessors and caching the discovered definitions in the map (ReadVariableRecursive in alg. 2). There are three cases we need to consider:

procedure WriteVariable(*variable*, *block*, *value*)*:*
  currentDef[*variable*][*block*] ← *value*

function ReadVariable(*variable*, *block*)*:*
  if *currentDef*[*variable*][*block*] *is set* **then**
    return *currentDef*[*variable*][*block*]
  end
  return ReadVariableRecursive(*variable, block*)

Algorithm 1: Implementation of local value numbering [23].

1. we do not know yet about all of the predecessor blocks (the block has not been *sealed* yet) – we create and return a temporary ϕ-node and queue it for later resolution,
2. the block has a single predecessor – we do not need to insert any ϕ-nodes, so we directly query the only predecessor for the definition,
3. the block has multiple predecessors – more than one definition of the variable can reach the current basic block and we need to *join* them using a ϕ-node.

The third case is the most interesting, because we cannot just recursively query the predecessors as that could lead to infinite recursion in case of loops in the CFG. Instead, before resolving the ϕ operands, we first create a blank ϕ-node and store it into the map as the current definition.

After we add operands to the ϕ-node, we try to recursively eliminate it in case it contains only one unique operand (not counting self-loops). When we know that no more predecessors will be added to a basic block, we *seal* it and resolve the queued incomplete ϕ-nodes (SealBlock).

This algorithm guarantees minimality for non-irreducible control flow (no jumps to the middle of a loop, for more formal definition see [6]). The authors designed an additional extension to construct minimal SSA for arbitrary control flow [23], but we will leave it as an exercise for the reader. To reach the goals of this thesis, constructing *some* SSA form is enough[9].

2.7 Optimizations

SSA construction is only one of many possible optimizations and transformations that we can do in the middleend (optimizer) of the compiler.

Most of the complexity of real world compilers (LLVM, GCC) lies here, because spending time with static analyses and optimizations once during compilation is better than during execution, which often happens repeatedly.

[9] and all control flow produced by our frontend is reducible, because tinyC does not have the `goto` statement

```
function ReadVariableRecursive(variable, block):
    if block ∉ sealedBlocks then
        val ← new φ-node(block)
        incompletePhis[block][variable] ← val
    else if |pred(block)| = 1 then
        val ← ReadVariable(variable, pred(b)[0])
    else
        val ← new φ-node(block)
        WriteVariable(block, variable, val)
        val ← AddPhiOperands(variable, val)
    end
    WriteVariable(block, variable, val)
    return val

function AddPhiOperands(variable, phi):
    foreach p ∈ pred(parentBlock(phi)) do
        appendPhiOperand(phi, ReadVariable(variable, p))
    end
    return TryRemoveTrivialPhi(phi)

function TryRemoveTrivialPhi(phi):
    uniqSet = uniquePhiOperands(phi) \ {phi}
    if uniqSet = ∅ ∨ |uniqSet| > 1 then
        return phi // unreachable or non-trivial
    uniqOp ← uniqSet[0]
    phiUsers ← insnUsers(phi)
    replace all uses of phi with uniqOp and remove phi
    foreach u ∈ phiUsers do
        if u is a φ-node ∧ u ≠ phi then
            // recursively optimize users
            TryRemoveTrivialPhi(u)
        end
    end
    return uniqOp

procedure SealBlock(block):
    foreach (variable, phi) ∈ incompletePhis[block] do
        AddPhiOperands(variable, phi)
    end
    sealedBlocks ← sealedBlocks ∪ block
```

Algorithm 2: Implementation of global value numbering and trivial ϕ-node removal [23].

All frontends and backends of a compiler can benefit from this work, because we optimize the generic IR.

Some often implemented optimizations include:

- **constant propagation**, which evaluates known constant expressions during compile time,
- **common subexpression elimination**, which simplifies expressions if their result is already known from an earlier point in the program,
- **strength reduction**, which replaces special cases of more general operations with a less powerful, but faster equivalents (for example multiplication by a power of two by left shift),
- **dead code elimination**, which deletes basic blocks that definitely will not be executed, and
- **function inlining**, which is an interprocedural optimization that replaces some call expressions with body of the called function, giving more context to further optimizations.

All those transformations have in common that they rewrite parts of the program with equivalent code, which has faster expected execution time. Figure 2.12 shows the LLVM IR of a small C program before and after optimization. Modern compilers are getting better at optimizing programs, but there is an inherent limit to static optimizations set by the computability theory. If we were to construct a fully optimizing compiler, which transforms a program to the smallest equivalent version, we could solve the halting problem and that is a contradiction. [19]

This thesis focuses more on the backend part of the compiler, so we will not go into any further detail and instead point the reader to [23] and [19], which describe practical implementations of the optimizations listed above.

```
define i32 @main() {
entry:
  %x = alloca i32
  %r = alloca i32
  %0 = call i32 @square(i32 5)
  %add = add nsw i32 1, %0
  store i32 %add, ptr %x
  store i32 0, ptr %r
  %1 = load i32, ptr %x
  %tobool = icmp ne i32 %1, 0
  br i1 %tobool, label %if.then, label %if.end

if.then:
  %call1 = call i32 @rand()
  %mul = mul nsw i32 %call1, 2
  store i32 %mul, ptr %r
  br label %if.end

if.end:
  %1 = load i32, ptr %r
  ret i32 %1
}
```

(a)

```
define i32 @main() {
entry:
  %0 = call i32 @rand()
  %mul = shl nsw i32 %0, 1
  ret i32 %mul
}
```

(b)

```
int square(int n) {
  return n * n;
}
int main() {
  int x = 1 + square(5), r = 0;
  if(x) r = rand() * 2;
  return r;
}
```

(c)

Figure 2.12: An example of a LLVM IR of a simple C program (a) before and (b) after optimizations (clang-16 -O1). Function inlining together with constant propagation have determined that `x` is always truthy, so the compiler could replace the `if` statement with its body. Strength reduction also replaced multiplication by two with a left shift. (c) shows the original C source code.

Chapter 3

Instruction Selection

Given a program in the intermediate representation, the goal of instruction selection is to implement a program with the same behavior using instructions of the target language. This is not a one-to-one mapping, because real machine instruction can often perform multiple operations at once (say an arithmetic operation, which stores the result in memory). On the other hand, because the IR can be used as an abstraction over multiple targets, some IR instructions have no direct mapping and have to be implemented by multiple target instructions or could even require some additional logic (for example the ϕ-node). [24, 19]

Instruction selection is generally regarded as a pattern matching problem – we substitute a pattern of IR instructions with a snippet of target code with the same semantics. This problem can be further divided into two subproblems: [24]

- **pattern matching**, during which we want to find all the candidate snippets of instructions which can be used to implement the piece of IR code (a basic block, function or the whole program), and
- **pattern selection**, which needs to select an optimal subset of candidates from those, while preserving the same semantics as the IR code.

Some techniques can combine those two into a single step, but oftentimes the pattern matching is similar and the main difference lies in the pattern selection phase [24]. We can choose from multiple criteria to minimize, but usually it is the execution time of the resulting code (which often correlates with smaller code size). More complex instruction selectors can take advantage of multiple addressing modes of the target ISA, which are also present in tiny86.

The main source of information for this chapter is the excellent book by Blindell [24], which describes a wide range of pattern matching and selection techniques in detail.

3.1 Overview of Instruction Selection Techniques

The first instruction selectors were implemented manually using ad-hoc algorithms. If performance and quality of the generated code is not a concern, we can usually implement a simple instruction selection with a recursive IR visitor. The ad-hoc instruction selectors are usually designed specifically for one architecture and reusing (re-targeting) it for a different ISA might even require a complete rewrite. On the other hand, if the algorithms were too general, the generated code may not be as efficient. This led to consensus on two main approaches: [24]

- **macro expansion**, where an expander takes the IR code and a list of target-specific macro definitions (each consisting of an IR instruction template and an associated action) and whenever it finds a match during traversal of the code, it executes the corresponding action, or
- **covering** of the IR code (tree, DAG or graph) with patterns of varying shapes.

3.2 Tree Covering

The main disadvantage of macro expansion is that we only expand a single IR instruction at a time. Macro expansion is also greedy – the expander stops after it finds a first match. While both of those limitations are usually to some extent alleviated by post processing the generated code with a peephole optimizer [24], we will now present an approach to instruction selection by tree covering.

In this section we assume that the program is decomposed into a set of IR trees and the compiler contains a built-in set of *tree patterns*, which model one or more target instructions. The goal of the instruction selector is to find a *covering* of the IR program (a set of pattern matches or *tree tiles*), where every IR instruction is covered by exactly one tile. We call such covering *optimal.*

As an example, we show a simple pattern set for two-address assembly language similar to tiny86 in table 3.1. The patterns store their result in the r_0 variable, which the instruction selector resolves to a fresh virtual register during code generation. Although it is small, this pattern set is still fairly repetitive, because the addition and multiplication operations are commutative. For that reason real world compilers generate the pattern set from a more concise machine description.

There are usually multiple possible optimal coverings for a given IR program. We denote *optimum* covering as the one with the lowest cost [19]. For example we will describe the options we have for covering the program expression $1+2*3$ (its tree representation is in fig. 3.1) using our simple pattern set. We could cover every IR node individually with the patterns (1) and (2), use

Target Instruction(s)	Tree Patterns
(1) `MOV` r_0`,` n	$\text{IImm}(n)$
(2) `MOV` r_0`,` r_1 `ADD/MUL` r_0`,` r_2	IAdd/UMul (r_1, r_2)
(3) `MOV` r_0`,` r_1 `ADD/MUL` r_0`,` v	IAdd/UMul (r_1, IImm(v)) IAdd/UMul (IImm(v), r_1)
(4) `LEA` r_0`,` `[`$r_1 + r_2 * v$`]`	IAdd (r_1, UMul (r_2, IImm(v))) IAdd (r_1, UMul (IImm(v), r_2)) IAdd (UMul (r_2, IImm(v)), r_1) IAdd (UMul (IImm(v), r_2), r_1)

Table 3.1: A simple pattern set for a tree-based IR and tiny86-like target language. The `LEA` instruction can be used to perform combined multiply-add operation in a single instruction.

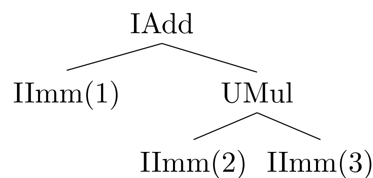


Figure 3.1: An intermediate representation of the expression $1 + 2 * 3$.

some combination of (1), (2) and (3), or we could use the largest `LEA` pattern (4) and two (1)s. It is the job of the pattern selector to find an optimum covering (or a close approximation thereof) and we will now describe some ways to implement it.

3.2.1 Maximal Munch

A simple algorithm which can find an optimal covering is called maximal munch [19]. It traverses the IR tree in a top-down manner and tries to match patterns from the largest to the smallest. The algorithm greedily selects the first matching pattern. The process is then repeated for every subtree rooted at a leaf of the pattern (the r_x variables in table 3.1). In the worst case, all patterns have to be tried for all instructions, which gives a complexity of $\mathcal{O}(n \cdot m)$, where n is the number of IR instructions and m is the size of the pattern set.

Thanks to the greedy nature of the algorithm, we can emit the target code during the same process before exiting a node. If two tiles have the same size (number of covered instructions), we can define a *cost* for each tile and use that as a second factor for sorting. Maximal munch is in this regard similar to macro expansion and does both pattern matching and selection at once.

If we represent the IR nodes using `case class`, we can implement a simple maximal munch instruction selector in Scala using pattern matching, like in listing 3.1. The implementation would look similarly in other languages.

```scala
def maxMunch(node: IrNode): VReg = {
  val r0 = freshReg(); node match {
    // patterns sorted from largest to smallest
    case IAdd(i1, UMul(i2, IImm(v))) =>
      val r1 = maxMunch(i1); val r2 = maxMunch(i2)
      emit(s"LEA $r0, [$r1 + $r2 * $v]")
    case IAdd(i1, IImm(v)) => val r1 = maxMunch(i1)
      emit(s"MOV $r0, $r1"); emit(s"ADD $r0, $v")
    case IAdd(i1, i2) =>
      val r1 = maxMunch(i1); val r2 = maxMunch(i2)
      emit(s"MOV $r0, $r1"); emit(s"ADD $r0, $r2")
    case IImm(v) => emit(s"MOV $r0, $v")
  }; return r0
}
```

Listing 3.1: An example maximal munch implementation in Scala for a subset of patterns from table 3.1

A related algorithm has been used in practice in the Portable C Compiler (PCC) developed for UNIX [2]. Johnson designed a machine description language to specify a list of rewrite rules. Each rule consists of a pattern (an IR instruction and subgoals), resource requirements (for example a certain type of register) and an action (a target code to emit verbatim).

The requirements and constraints are encoded as an integer called a *cookie* (a bitmask). By *or*-ing multiple of these cookies together, we can create more general rewrite rules and make the definition more concise. In listing 3.2 we

```
ASG PLUS,   INAREG,
            SAREG,      TINT,
            SNAME,      TINT,
                        0,      RLEFT,
                        "       add     AL,AR\n",
...
ASG OPSIMP, INAREG|FORCC,
            SAREG,      TINT|TUNSIGNED|TPOINT,
            SAREG|SNAME|SOREG|SCON, TINT|TUNSIGNED|TPOINT,
                        0,      RLEFT|RESCC,
                        "       OI      AL,AR\n",
```

Listing 3.2: An example of two patterns from a PCC machine description. The first pattern matches a `+=` operator if the result should be stored in the A register, the first operand is an integer argument stored in A register and the second operand is a `NAME` node. If the match succeeds, the compiler emits the assembly code (after expanding the `AL` and `AR` macros) and replaces the matched IR subtree with the left subtree (`RLEFT`). The second pattern is used for matching `+`, `-`, `|`, `&` and `^` arithmetic operators and uses the `OI` macro and `|` operator for more concise notation. [2]

show two sample rewrite rules, where the second one is generalized using the *or* operator.

The PCC supports only unary and binary patterns with a maximum height of 1, which puts it closer to macro expansion, although the IR is still represented as a set of trees. If the compiler does not find any match for a piece of IR code, it attempts to match it again after using some heuristics (for example rewriting `+=` with `=` and `+`).

3.2.2 Dynamic Programming

Maximal munch finds an optimal covering (every instruction is covered by exactly one matched pattern), but it is not always optimum. This happens in cases such as if the pattern has some additional requirements on the result of its leaves (e.g. a specific register type). Because the pattern matching works from the top down, the algorithm does not know if the requirements can be satisfied and that can force it later to do suboptimal choices (e.g. an additional move between register type). To find an optimum covering, we can use pure decomposition and work from the bottom up – we construct an optimal solution for the whole problem from optimal solutions of its subproblems. [19, 25]

The algorithm works by recursively assigning a cost to each node (IR instruction) in the program tree. We first process all children of node n in a

bottom-up fashion and find all matching patterns rooted at n (we can delegate this to a separate pattern matching algorithm). The cost of a match is sum of its own cost (defined as part of the pattern) and the costs of its leaves. From all of those matches we select one with the lowest cost and assign it to the node n. We avoid recomputing the optimal costs of subtrees by storing them in a table, see pseudocode in alg. 3.

```
bestChoiceForNode : Node → (Tile × int)
tileMap : Node → Tile
procedure BottomUpDp(n)
    foreach c ∈ children(n) do
        BottomUpDp(c)
    end
    bestChoiceForNode[n].cost ← ∞
    foreach t ∈ matchingTiles(n) do
        totalCost ← cost(t)
        foreach l ∈ tileLeaves(t) do
            totalCost ← totalCost + bestChoiceForNode[l].cost
        end
        if totalCost < bestChoiceForNode[n].cost then
            bestChoiceForNode[n].cost ← totalCost
            bestChoiceForNode[n].tile ← t
        end
    end
procedure TopDownSelect(n)
    t ← bestChoiceForNode[n].tile
    tileMap[n] ← t
    foreach l ∈ tileLeaves(t) do
        TopDownSelect(l)
    end
```

Algorithm 3: A pseudocode of a simple bottom-up instruction selector utilizing dynamic programming.

After the recursion ends, we have computed minimum costs and matches for the entire IR tree. We now perform a top-down pass starting at the root and for each visited node n, we perform the action specified by the match (usually code emission) and recurse into the *leaves* of the match (we skip the inner nodes).

The dynamic programming technique can be easily extended to support multiple kinds of patterns (e.g. multiple register types). We can store the minimum cost and corresponding tile for each pattern kind and select the tile with the lowest cost of all kinds at the root.

3.2.3 Extending Tree Covering to DAGs

Both maximal munch and dynamic programming work with IR represented as a set of program trees. However, tree-based intermediate representations cannot represent common subexpressions without explicitly using temporary variables. When we lift this restriction, we get a structure called directed acyclic graph (DAG), which is one of the kinds of intermediate representations we have described in chapter 2. In this section we will limit ourselves to tree-shaped patterns, techniques for matching more general DAG patterns can be found in [24]. Still, matching trees on DAGs is a NP-complete problem [26] and the techniques used by compilers often sacrifice optimality to shorten compile times.

One way to tackle the problem is by decomposing the DAG into a set of trees by dealing with the shared nodes. There are two general ways to do this, which can be combined on a per-node basis: [24, 27]

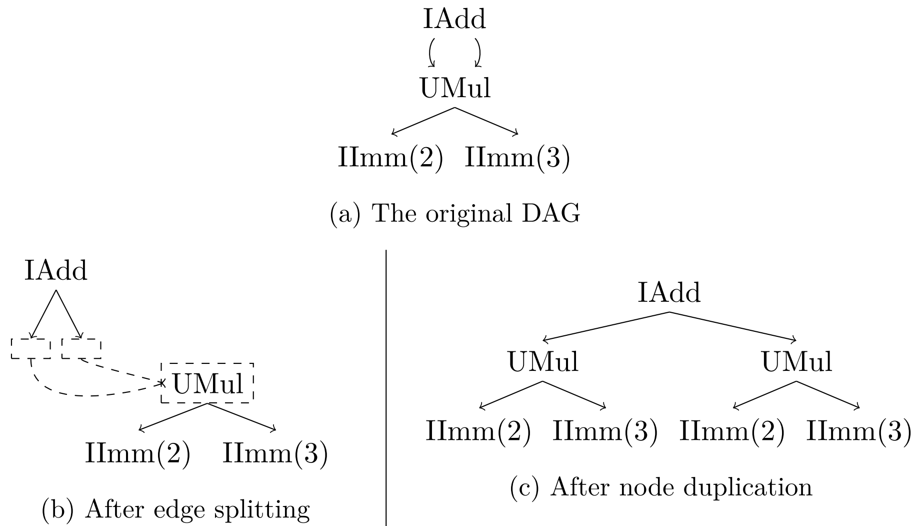
- we can **split the edges** involving a shared node n and make an implicit connection between the copies of n by storing it in a shared temporary (register or memory location), or
- we can **duplicate the node** for each of its uses.

Both of those solutions compromise on the quality of the generated code. Edge splitting computes the result of the shared node only once, but it relies on the register allocator to optimize accesses to the temporary location. It also forces the shared node to be always at the root of a pattern, which limits the use of more complex patterns. Node duplication causes re-evaluation of the same code multiple times, which leads to larger code and longer execution time. In addition, we cannot duplicate nodes with side effects. We illustrate a simple example of both edge splitting and node duplication in fig. 3.2.

In real implementations, we try to find balance between edges splitting and node duplication. One such algorithm was developed by Fauth et al. [28] for use in the Common Bus Compiler. The instruction selector used a heuristic, which tries to decompose and cover the whole IR program using both approaches and finally selects the one that is deemed better in terms of code size and estimated execution time.

3.2.4 NOLTIS

NOLTIS is a near-optimal, linear time instruction selection algorithm for DAG expressions developed by Koes and Goldstein [27]. It is an extension of the dynamic programming approach, which can switch between edge splitting and node duplication on a per-node basis. The algorithm first performs a bottom-up DP pass, which treats the DAG like a tree (thus in a way simulates decomposition by node duplication). It then for every shared node n decides whether splitting the related edges would result in lowering of the overall cost



(a) The original DAG

(b) After edge splitting

(c) After node duplication

Figure 3.2: An example showing the effect of (b) edge splitting and (c) node duplication on a common subexpression. After edge splitting the UMul is evaluated only once and its result is stored in a register.

of the covering. It does this by comparing *overlapCost* (the cost of letting the tiles covering n overlap) and *cseCost* (the cost of splitting the tiles at n). If the *cseCost* is lower than *overlapCost*, the node is added to a set of fixed nodes. After making this decision for all nodes, it does another bottom-up DP pass, but the fixed nodes can be only matched by a root of a pattern. Lastly, it emits the target code in a top-down pass.

The algorithm depends on several assumptions about the set of patterns, mainly that it is always possible to cut a tile at a shared node without affecting the existence of a solution [27].

Koes and Goldstein implemented [27] the NOLTIS algorithm in LLVM together with two additional variants of DP with only edge-splitting (*cse-all*) and only node-duplication (*cse-none*). They run an experiment on a suite of SPEC CPU2006 benchmarks comparing the overall tiling cost of the program, where NOLTIS performed the best with cse-none in the second place.

3.2.5 Common Target-Specific Problems

In this section we list some commonly encountered problems when implementing a tree-covering instruction selector and their solutions. Although most modern architectures are designed as RISC [19], many machines used today contain CPUs from the Intel 8086 family, which are examples of the complex instruction set computer paradigm. A typical instruction taken from a CISC ISA supports multiple addressing modes and can perform more complex operations, that would take multiple instructions on a RISC machine.

3.2.6 Limited Number of Registers

Some older CISC ISA contain a very limited number of registers (e.g., 6 on 32-bit x86). This is more of a concern for the register allocator, the instruction selector can still allocate fresh virtual registers (temporaries) as required. In more complex compilers, the instruction selection and register allocation can work in tandem and for example select different patterns in case of register pressure (register-sensitive instruction selection) [24].

Through the complex CISC addressing modes we can also often work with a value directly in memory without loading it into a register. With an appropriate feedback from the register allocator, using those modes could be in some cases beneficial even though memory is slower than accessing the register file. We can also add an additional scheduling pass to reorder instructions so less registers are required at once.

3.2.7 Multiple Output Instructions

Some instructions have multiple outputs (for example an arithmetic operation that sets flags in the ALU or a `DIV` instruction that performs division and remainder at once into two destination registers) or side effects (such as auto-increment of address after memory fetch). We cannot directly model those using tree patterns with a single root, so we have a few options: [19]

- ignore those instructions – they often take the same amount of ticks as a sequence of simpler instructions,
- try to match them in ad-hoc way (possibly using a peepholer pass),
- use a different algorithm for matching DAG patterns.

A possible solution for matching DAG patterns without introducing a completely new algorithm was proposed by Arnold in [29]. His design permits emitting code from nodes other than the root of a pattern, enabling a DAG pattern to be decomposed into multiple tree patterns (one for each output). After matching those smaller trees, the algorithm tries to combine them back into the original pattern DAGs.

3.3 Phi Node Elimination

The limitation of DAG intermediate representation is that it is unable to represent loops in the data flow. In section 2.4 we have described a notion of SSA intermediate representation which uses ϕ-nodes to facilitate loops. These are virtual instructions, which merge data flow from multiple predecessor blocks and they have no direct equivalent in a target ISA.

Although there are methods for covering a generic graph [24], in this thesis we will limit ourselves to a simple adaptation of DAG covering. We can

break the loops by removing incoming edges of every ϕ-node. If some of its predecessors are shared, we must ensure that their results are available at the time of code emission (they are covered by roots of patterns). Then during code emission, we perform a transformation called SSA destruction, where we replace the ϕ-nodes with `MOV` instructions and append additional `MOV`s at the end of some basic blocks. Before we can describe SSA destruction in more detail, we need to define live range of a variable.

Definition 8. A variable in a SSA program is *live* at all program points between each of its uses and its (single) definition. A set of program points where a variable is live is the *live range* of the variable. Two live ranges *interfere* if there is a program point common to both of those ranges. [30]

When we say variable in the context of instruction selection, we usually mean a virtual register. Those virtual registers are allocated freely during code emission and we usually assign to them only once. It is the responsibility of a register allocator to map those virtual registers to physical ones in a later phase of compilation.

One could come up with a naive method of SSA elimination by simply merging the inputs and output of a ϕ-node into the same variable, but that can affect correctness of the resulting code. See fig. 3.3a showing an example CFG of a code with a simple `if` statement. After we eliminate the ϕ-node by merging x, y and z in fig. 3.3b, the program becomes obviously incorrect. We can fix this by emitting a copy into z at the end of b_1 and b_2 (fig. 3.3c), but that may still result in incorrect programs. We will show two examples of such issues, the *lost-copy* problem and the *swap problem.* [30]

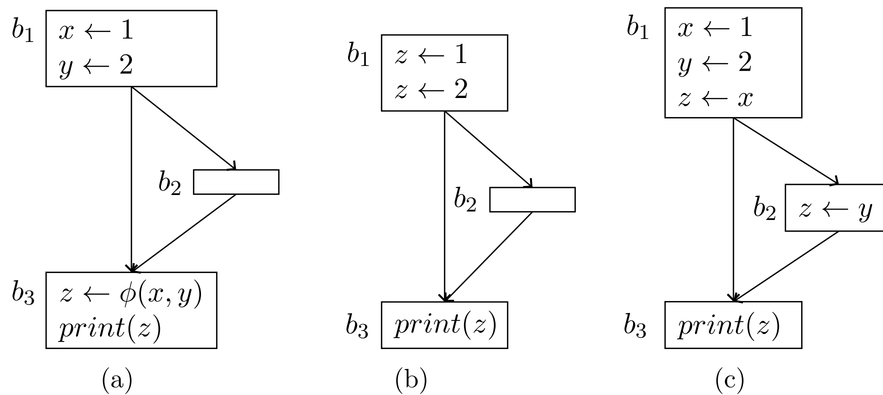


Figure 3.3: An example of a shortcoming of naive ϕ-node elimination. (a) shows the original CFG, (b) shows the incorrect result of merging the x, y and z variables, which was corrected in (c) by inserting copy statements.

In the next example we will use the term *critical edge.*

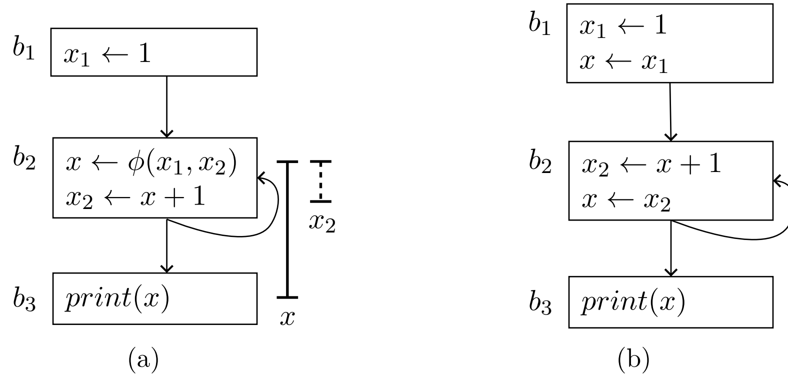


Figure 3.4: An example demonstrating the lost-copy problem when we hoist a copy across a critical edge. The program in (b) incorrectly prints the value of x from the last instead of the penultimate iteration.

Definition 9. An edge $u \to v$ is *critical* if u has multiple successors and v has more than one predecessor.

The lost-copy problem can occur when we *hoist* a copy across a critical edge in the CFG, which creates an interference between an argument of a ϕ-node and its result [30]. We demonstrate this issue in fig. 3.4, where the program after ϕ-node elimination prints the value of x from the last (n-th) iteration instead of the previous ($n - 1$-th). We can see in (a) that the live ranges of x (solid) and x_2 (dashed) interfere.

The swap problem is another example of common issue caused by incorrect SSA destruction. The correct semantic is that all ϕ-nodes at the beginning of a block are executed at once and their results are assigned simultaneously. However by inserting the copy statements we serialize the assignments as a fixed order sequence. We have included an example of this problem in fig. 3.5, where the values of x and y should be swapped in b_2. [30]

A simple solution of both the lost-copy and swap problems is to copy the arguments into a fresh temporary variable instead of referencing the result of the ϕ-node directly. This could be further refined by minimizing the number of copy instructions (refer to [30] for an algorithm to do just that), but to keep things simple we delegate this work to the register allocator. For completeness, we illustrate a fixed version of both of the previous examples in fig. 3.6.

3.4 Macro Expansion

Before tree covering was widespread, retargetable compilers used another instruction selection technique called macro expansion. Modern compilers such as GCC still use this method, but combined with a powerful peephole optimizer [20].

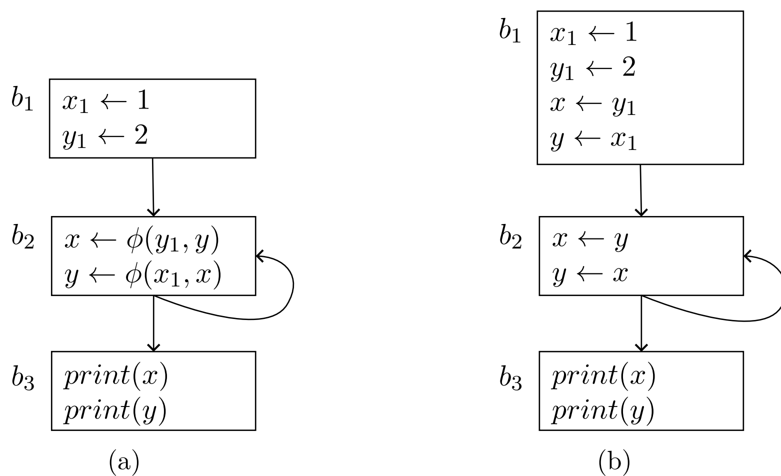


Figure 3.5: An example of the swap problem, the program in (b) is incorrect.

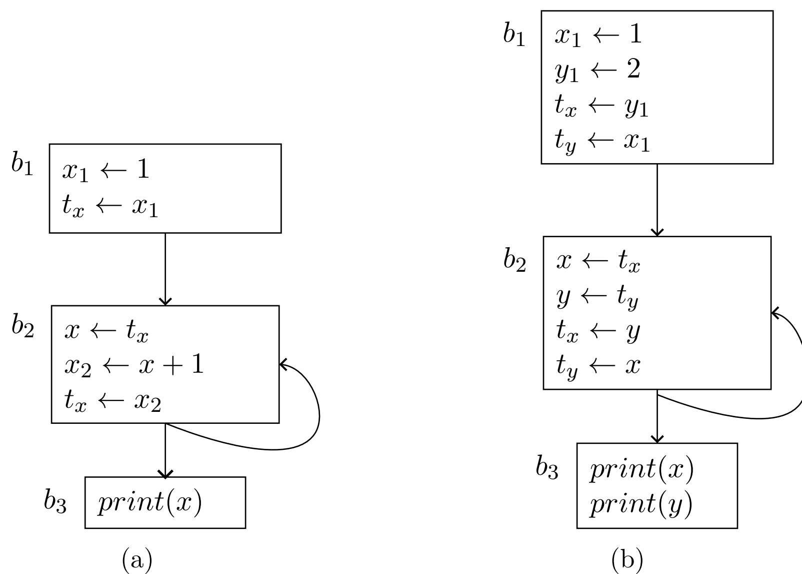


Figure 3.6: A corrected version of (a) the lost copy and (b) swap problem examples by introducing temporary variables t_x and t_y.

The macro expansion itself scans the linearized representation of the IR and matches every instruction against a defined set of macros. Whenever we find a match, we emit the corresponding target code. All macro-expanding instruction selectors studied by Blindell were greedy and stop expanding on a first match [24].

In early implementations, the macros were written by hand, but that has proven to be tedious and prone to errors, because sometimes the macros also dealt with register allocation and had to keep track of locations of their operands [24]. Later systems (such as DMACS [31]) generate the set of macros from a simpler machine description languages.

The main advantage of macro expansion over tree covering is its speed. For that reason it is widely used in JIT compilers, which generate the target code at runtime for sections that are recognized by a profiler as frequently executed. Even naively generated machine code can be orders of magnitude faster than relying on an interpreter.

3.4.1 Peephole Optimization

The code generated by macro expansion can be further improved by peephole optimization. The optimizer slides a small window (a peephole) across the generated assembly code and attempts to replace the visible adjacent instructions with a single instruction while preserving the program behavior.

Like with macro expansion, the earliest implementations of a peephole optimizer contained a hand-crafted list of patterns and their replacements. A better approach, required for more complex ISAs is to describe observable behavior of instructions (including side effects like setting ALU flags) using register transfer lists (RTLs) and let the compiler generate the patterns automatically [32].

Davidson and Fraser [32] implemented a peephole optimizer that utilizes RTLs and optimizes a program in two passes. In the first pass, the peephole optimizer processes the program backwards and determines observable effects of each assembly instruction. In the second pass, it looks at adjacent pairs of instructions and compares their observable effects (sum of RTLs) with the instructions available in the ISA. If it finds a match, it replaces the pair with the new instruction.

Modern compilers such as GCC use [20] a similar approach also designed by Davidson and Fraser [33]. The difference from the previous technique is that we generate RTLs directly from the intermediate code in a process called *expansion*. The *combiner* then performs the peephole optimization on RTLs and a final *assigner* pass emits the assembly code.

Chapter **4**

Register Allocation and Assignment

After instruction selection, we are left with a listing of target instructions grouped into basic blocks and functions. But this code cannot be executed directly by the target, because the instructions use *virtual registers* (*temporaries*) to store operands and results. Using unlimited number of temporaries had the convenience that we did not need to track their lifetime and check for collisions, but we now need to rewrite the program to reference only the limited number of *physical registers* available in the target without changing its semantics. We divide this process into two parts: [6]

- **register allocation**, which decides what subset of temporaries will be stored in registers (we *spill* the rest into the slower memory), and
- **register assignment**, which for each of the temporaries stored in a register assigns its exact location (register number).

We further categorize allocators as either *local*, which perform register allocation for each basic block separately, or *global*, which take a whole function into account [6]. We will first describe a simple algorithm for simultaneous local register allocation and assignment, which can be used as a stopgap solution before introducing two more advanced global register allocation algorithms.

4.1 A Simple Local Register Allocator

We can do a basic form of register allocation even during code generation by loading referenced temporaries from memory into registers just in time before each instruction. At the end of a basic block or if all registers are occupied with dirty values, we spill the dirty (modified) temporaries back into memory. Bellard uses this approach in the Tiny C Compiler [21] and Aho et al. also describe it in their book [6].

The algorithm needs to track what is stored in each of the physical registers and where is each temporary stored. For that it uses two descriptors:

- **a register descriptor** keeps track of the currently present temporary in each physical register and whether it may have been modified, and
- **an address descriptor** contains for each temporary a set of locations it is currently present in. This can be a physical register number, a location on the stack or both.

The algorithm is simple to implement but suboptimal, because it keeps all temporaries in memory between basic blocks, which has a significant performance penalty (for example in loops). A better solution is to use a global register allocator, which tracks *liveness* of temporaries across basic blocks and assigns registers to temporaries in a way that they do not interfere.

4.2 Liveness Analysis

An informal definition of liveness is that a temporary is live at a program point, iff its current value will be read later. Having liveness information is important for global register allocation, because if two temporaries are both live at some program point, they cannot share a register.

We will describe methods to perform liveness analysis statically on the instruction CFG of the program, which is a conservative approximation of dynamic (real) liveness, but can be computed in bounded time during compilation. The analysis determines if a variable *may* be live, so it belongs to a class of *may* dataflow analyses. To compute liveness, the compiler has to know what temporaries may be read or written to by an instruction.

Definition 10. An instruction *defines* a temporary when it may change its value. An instruction *uses* a temporary, when it may access its value. $\mathrm{def}(t)$ is the set of CFG nodes (instructions) that define temporary t. $\mathrm{def}(n)$ is the set of temporaries defined by node n. We define $\mathrm{use}(t)$ and $\mathrm{use}(n)$ similarly.

Definition 11. The temporary t is live at an edge e of the CFG if there exists a directed path from $u \in \mathrm{def}(t)$ to $v \in \mathrm{use}(t)$ that includes e and does not go through any other node from $\mathrm{def}(t)$. A temporary is *live-in* at a node when it is live-in at any of the incoming edges of that node. A temporary is *live-out* at a node iff it is live at any of the outgoing edges of that node.

Because we are interested in the uses of temporaries in the future, we analyze liveness by going backwards against the direction of edges of the CFG. The CFG may contain loops, so we will compute live-in and live-out sets for each node iteratively using the following dataflow equations:

$$in[n] = (out[n] \setminus \text{def}(n)) \cup \text{use}(n) \tag{4.1}$$

$$out[n] = \bigcup_{s \in \text{succ}(n)} in[s] \tag{4.2}$$

Going backwards through a node n first stops (kills) the liveness of all temporaries defined by n and then starts liveness of all variables used by n. If a node both defines and uses the same temporary, it still starts its live range, because the node has to have access to the old value. Equation (4.2) is the definition of a *live-out* temporary (def. 11).

We can solve these equations using a naive iterative algorithm shown in alg. 4. The solver starts with empty sets of live temporaries for each node and updates them for each node using the equations above until they no longer change. The computation is monotonic ($in'[n] \supseteq in[n]$ and $out'[n] \supseteq out[n]$), so the solver halts after a finite number of iterations. When it does, we say that it has reached a *fixed point.*

The equations can have multiple solutions, but because we start with empty sets, the solver always returns the least fixed point, which is the most precise solution. [19]

Input: Control-flow graph of instructions G
Output: A set of live-in ($in[n]$) and live-out ($out[n]$) temporaries for each node n of G

foreach $n \in \text{nodes}(G)$ **do**
  $in[n] \leftarrow \emptyset$; $out[n] \leftarrow \emptyset$
end
repeat
  foreach $n \in \text{nodes}(G)$ **do**
    $in'[n] \leftarrow in[n]$; $out'[n] \leftarrow out[n]$
    $out[n] \leftarrow \bigcup_{s \in \text{succ}(n)} in[s]$
    $in[n] \leftarrow \text{use}(n) \cup (out[n] \setminus \text{def}(n))$
  end
until $in'[n] = in[n] \wedge out'[n] = out[n] \ \forall n \in \text{nodes}(G)$

Algorithm 4: A simple iterative solver for liveness analysis.

We demonstrate steps of this algorithm on CFG from fig. 4.1 in table 4.1. The algorithm found the fixed point after 4 iterations and we can read the live-in and live-out variables for each node in the last column.

Another, less imperative way to compute liveness is by representing the program state as a lattice and reformulating the dataflow equations as a transfer function.

Definition 12. A *partial order* is a set S and a reflexive, transitive and anti-symmetric binary relation $\sqsubseteq$. A *lattice* is a partial order in which we

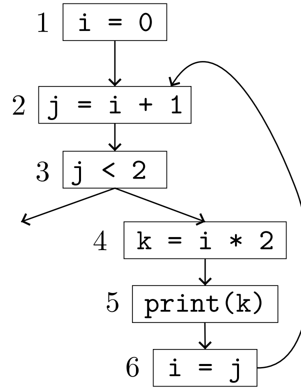


Figure 4.1: Example CFG of a program for demonstrating liveness analysis.

b	def	use	out_1	in_1	out_2	in_2	out_3	in_3	out_4	in_4	out_5	in_5
1	i				i		i		i		i	
2	j	i		i	j	i	i,j	i	i,j	i	j,i	i
3		j		j	i	i,j	i	i,j	j,i	j,i	j,i	j,i
4	k	i		i	k	i	j,k	j,i	j,k	j,i	j,k	j,i
5		k		k	j	j,k	j	j,k	j	j,k	j	j,k
6	i	j	i	j	i	j	i	j	i	j	i	j

Table 4.1: Steps of the iterative solver for the CFG from fig. 4.1.

can compute least upper bound ($x \sqcup y$) and greatest lower bound ($x \sqcap y$) for every $x, y \in S$. A *complete lattice* is a lattice, where we can compute least upper bound ($\bigsqcup X$) and greatest lower bound ($\bigsqcap X$) for every subset $X \subseteq S$. Every complete lattice has a unique largest element $\top = \bigsqcup S$ (top) and unique smallest element $\bot = \bigsqcap S$ (bottom). For complete definition see [1].

We can visualize a complete lattice as a diamond-shaped Hasse diagram with a finite height, where the top and bottom levels contain exactly one element ($\top$, resp. $\bot$). We show two examples in fig. 4.2.

We will use a powerset lattice to store the live variables for a CFG node.

Note 1 (Powerset Lattice). If A is a set, then $(\mathcal{P}(A), \subseteq)$, where $\bot = \emptyset$, $\top = A$, $x \sqcup y = x \cup y$, and $x \sqcap y = x \cap y$ is a complete lattice.

We will use two lattices to compute liveness of variables in the program. A *node state lattice* is a powerset lattice describing the live-in variables of a node. Using it we node-wise define a *program state lattice*, which is a map lattice that holds a node state lattice for each node of the program [1].

We define a monotone function $t_u(s)$, which given a program state s com-

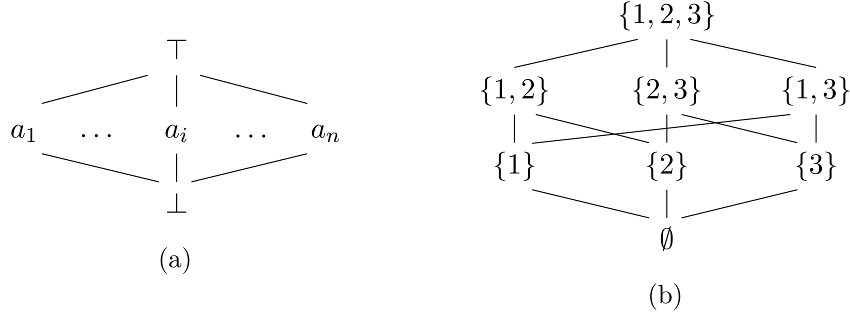


Figure 4.2: Hasse diagram of (a) a flat lattice and (b) a powerset lattice.

putes the new live-in variables for node u ($t_u(s) \subseteq \text{variables}(G)$).

$$join(u, s) = \bigcup_{v \in \text{succ}(u)} s[v]$$
$$t_u(s) = (join(u, s) \setminus \text{def}(u)) \cup \text{use}(u)$$

Using $t_u(s)$, we can node-wise define a monotone transfer function $f(s)$ for the whole program state (a map $\text{nodes}(G) \to \mathcal{P}(\text{variables}(G))$).

$$f(s) = [u \to t_u(s) \mid u \in \text{nodes}(G)]$$

Due to Kleene's theorem, the least fixed point $lfp(f)$ is defined as

$$lfp(f) = \bigsqcup_{i \geq 0} f^i(\bot).$$

Because f is monotone, $\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots$ and we are searching for the first k, where $f^k(\bot) = f^{k+1}(\bot)$, which is our $lfp(f)$ [1]. We can write this in pseudocode as alg. 5.

```
procedure NaiveFixedPoint(f)
  x ← ⊥
  while x ≠ f(x) do
    x ← f(x)
  end
  return x
```

Algorithm 5: A naive iterative algorithm to find the least fixed point of f starting at $\bot$ by computing $f^k(\bot) = f(f^{k-1}(\bot)), k \in \mathbb{N}$ until the result no longer changes [1].

We can further improve this solver by using worklists to avoid recomputing states of nodes that definitely did not change [1].

Although the lattice theory may seem daunting at first, it allows us to construct a general *monotone framework*, which can be used for many other

dataflow analyses just by providing an appropriate node transfer function $t_u(s)$. For example: [1]

- **sign analysis**, which determines the signs (+, −, 0) of expressions,
- **constant propagation analysis**, which calculates the subset of expressions that always evaluate to a constant value – we then use the information for the optimization mentioned in section 2.7, or
- **available expression analysis**, which determines what expressions have already been computed earlier in the execution of a program.

4.3 Register Allocation by Graph Coloring

We can reformulate global register allocation as a graph coloring problem. Using the liveness information of temporaries, we construct an *interference graph*, where nodes correspond to temporaries and edges are present between temporaries that cannot share a register – there exists a program point where both of them are live (with one exception we mention later). We then color each of the nodes in the graph with one of K colors, where K is the number of physical registers in the target. Thanks to the property of graph coloring, where two adjacent nodes cannot have the same color, we get a valid register assignment. This method of register allocation was first proposed by Chaitin et al. [34] and it is now used by GCC [20].

If the graph is not colorable, we choose some temporaries to spill into memory – this splits their live ranges into very short segments, which simplifies the interference graph and makes the coloring easier. We iteratively do this until we find a valid solution.

The one exception where two temporaries t_d and t_s do not interfere even though they are both live at a single program point is when they are only used together as part of one or more move instructions $t_d \leftarrow t_s$. We take this one step further and modify the coloring algorithm so that is tries to assign the same color to both t_d and t_s, allowing the move to be eliminated.

Determining if a graph can be colored with K colors is a NP-complete problem, so compilers use algorithms based on heuristics.

Chaitin et al. [34] were the first to use graph coloring for register allocation with a heuristic called coloring by simplification. Before the coloring itself, their algorithm aggressively coalesced all move instructions. When the graph was not K-colorable, it chose registers to spill randomly.

Briggs et al. [35] improved the coloring heuristics by delaying spilling until the color assignment phase (optimistic coloring). To determine what node to spill it calculates spill cost (determined by number of defines and uses of a temporary) and selects node with the lowest cost. The algorithm uses conservative coalescing, but does it all at once before simplification.

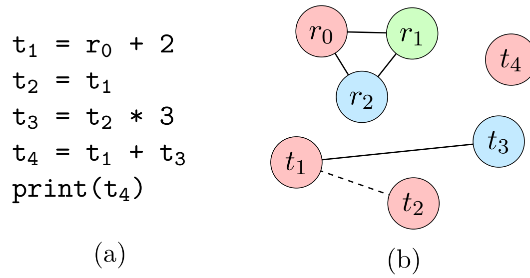


Figure 4.3: A sample source code and a corresponding colored interference graph with three precolored physical registers r_1, r_2 and r_3. t_1 and t_2 do not interfere, because they are related only by a move.

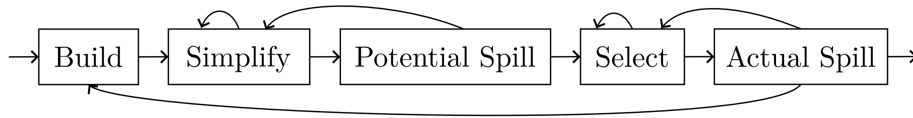


Figure 4.4: Diagram of phases of the graph coloring register allocation algorithm

Finally, George and Appel [36] proposed an algorithm that interleaves conservative coalescing with simplification, which allows more moves to be coalesced than in the previous algorithm. This is the version of graph coloring register allocation that Appel also described in his book [19] and which we describe in the rest of this section.

4.3.1 Coloring by Simplification

First, we describe a simplified version of the algorithm without move coalescing and then extend it to the full version. The optimistic coloring algorithm works in 5 phases, illustrated in fig. 4.4: **Build**, **Simplify**, **Potential Spill**, **Select** and **Actual Spill**.

Build We construct an interference graph from results of the liveness analysis. For performance reasons, we perform the analysis on basic blocks and compute liveness of individual instructions during construction of the interference graph. We add an edge to the graph for every pair of temporaries that are live at the same time at some point in the program (excluding the register-register moves as described above). The graph also contains a precolored node for each of the physical registers.

Simplify We repeatedly search for some node v of non-significant degree ($deg(v) < K$), remove it from the graph and push it onto a stack. We can show that if $G-v$ is colorable with K colors, then there also exists coloring for G. Even if all of the $deg(v) < K$ neighbors of v have been assigned different

colors, there will still be one left for v. After the graph has been simplified by removing v, the degree of its original neighbors decreases by one and it may now be possible to simplify them.

Potential Spill After the simplify phase, we are either left with an empty graph and we jump straight into the select phase, or there are only nodes of significant degree. In the latter case, we select one of the remaining temporaries to spill, remove it from the graph and push it onto the simplification stack.

Select We rebuild the graph by repeatedly popping nodes from the stack and assigning them colors. For each node we find a color that has not been assigned to any of its neighbors yet and assign it to the node. This will succeed for all nodes pushed onto the stack during the simplify phase, but it may fail for a node added in the spill phase if all of its neighbors have already been assigned at least K different colors. In that case we add the node to a list of spilled nodes and continue the select phase to potentially identify more spilled nodes.

Actual Spill If we were not able to color all nodes in the select phase, we rewrite the program with the spilled temporaries represented in memory. We emit an instruction to store them into memory after each *def* and a load instruction before each use. We must use fresh temporaries to hold the freshly loaded values, so the live ranges are short. After we have rewritten the program, we repeat the whole process until no additional spills have been requested. Only few iterations are needed in practice [6].

4.3.2 Move Coalescing

The typical code emitted by the instruction selector contains many move instructions ($t_d \leftarrow t_s$). If t_d and t_s do not interfere, we can *coalesce* them into a single node $t_d t_s$, which causes both of them to have the same color and allows the move to be deleted. We will now define two *conservative* coalescing strategies – they allow coalescing two nodes only if it does not affect colorability of the graph.

Theorem 1 (Briggs)**.** If G is colorable, then G' created by coalescing nodes u and v into uv is also colorable if the node uv will have fewer than K neighbors with degree $\geq K$.

Proof. After coalescing, we can simplify all the insignificant-degree neighbors of uv, which leaves uv with at most $K-1$ neighbors and thus we can simplify uv too. If G was colorable, this new simplified subgraph of G must be too. □

Theorem 2 (George)**.** If G is colorable, then G' created by coalescing nodes u and v into uv is also colorable if for all neighbors t of u it holds that t either has insignificant degree or t interferes with v.

Proof. With or without coalescing, we can simplify all insignificant-degree neighbors of u and we are left only with the ones that were interfering with v. After coalescing, the node uv is equivalent to v and thus the graph must be as easy to color as G. □

We interleave coalescing with simplification, because if the degrees of nodes are lower, the conservative criteria have a higher change of succeeding. This is one of the improvements made by George and Appel [36].

To handle move coalescing, we extend the algorithm with two new phases – **Coalesce** and **Freeze**.

Build Same as before, except we now take note of move instructions and move-related nodes.

Simplify We remove all non-move-related nodes with insignificant degree and push them onto the stack.

Coalesce If the graph can no longer be simplified, we check whether any of the moves in the graph can be coalesced using the conservative criteria. If the resulting node is no longer move-related, we can resume simplification.

Freeze If all insignificant-degree nodes in the graph are move-related but we cannot coalesce any more moves, we choose some move and *freeze* it – we no longer consider it as candidate for coalescing and remove it from the graph. This may cause its operands to become non-move-related and thus possible to simplify.

Potential Spill Same as before – all nodes in the graph have significant degree, so we choose one for potential spilling, push it onto the stack and remove it from the graph.

Select Same as before – we pop nodes from the stack and assign colors.

Actual Spill If there were any spilled nodes, we rewrite the program with spilled nodes as before and restart the algorithm.

4.3.3 The Complete Algorithm

Appel describes an implementation of this graph coloring algorithm with conservative move coalescing in [19]. It uses worklists to prevent quadratic time complexity otherwise caused by evaluating the coalescing criterion for every move at all times. Before and after each step (Simplify(), Coalesce(), Freeze(), SelectSpill(), AssignColors()) a node (temporary) is always in exactly one of the following data structures:

- **precolored** – a set of precolored temporaries (physical registers), whose *color*[n] is already before running the algorithm,
- **initial** – a set that contains non-precolored nodes ready for distribution by MakeWorklist() into *spillWorklist*, *freezeWorklist* and *simplifyWorklist*,
- **simplifyWorklist** – a set of non-move-related nodes with *degree*[n] $< K$ or nodes selected for optimistic spilling, ready to be removed from the graph by Simplify(),
- **freezeWorklist** – a set of move-related nodes with *degree*[n] $< K$,
- **spillWorklist** – a set of nodes considered for spilling,
- **coalescedNodes** – a set of nodes removed from the graph by coalescing with other node,
- **selectStack** – a stack of simplified nodes waiting to be assigned color by AssignColors(),
- **coloredNodes** – a set of non-precolored nodes that have been popped from *selectStack* and successfully assigned a color into *color*[n], and
- **spilledNodes** – a set of nodes to be spilled by RewriteProgram(). If this set is non-empty, then the graph was not colorable.

The algorithm uses a similar collection of disjunct sets to track move instructions. We have to be aware that there may be multiple moves with the same source and destination in a program, so we have to compare them by reference. Each move starts in *worklistMoves*, the rest has the following purposes:

- **worklistMoves** – a worklist of moves queued for coalescing by Coalesce(),
- **activeMoves** – moves which cannot be coalesced, because they failed the conservative criterion and thus could render the graph uncolorable,
- **frozenMoves** – moves that were too long in *activeMoves* (we have simplified and coalesced every other node and move that we could), so we have given up on coalescing them,

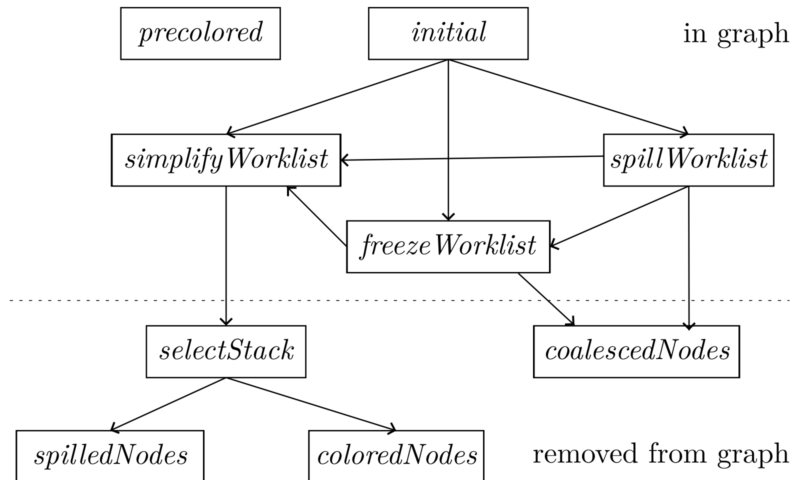


Figure 4.5: Lifecycle diagram of a node. A node starts either in the *precolored* or *initial* set depending on whether it represents a physical or virtual register. We treat nodes in sets below the line as if they have been removed from the graph (they are not included in Adjacent(n) and $degree[n]$).

- **coalescedMoves** – moves that have been coalesced ($alias[n]$ contains the sibling), and
- **constrainedMoves** – moves with interfering source and destination, which cannot be coalesced.

We only consider the moves in *worklistMoves* and *activeMoves* as "active" (for RelatedMoves(n)). We do not include the full pseudocode in this thesis and refer the reader to the original sources ([19] or [36]).

4.3.4 Expressing Constraints using Interference

An advantage of graph coloring is that we can easily use it to express additional constraints on registers by adding edges to the interference graph. For example we can force a temporary to be colored with one of a specific subset of physical registers by making it interfere with all of the others.

We can also modify the def and use sets of `CALL` and `RET` instructions to let the allocator automatically backup and restore caller- and callee-save registers.

If we make every live temporary at a `CALL` instruction interfere with all of the caller-save registers, we implicitly instruct the register allocator to either color each of those temporaries with a non-caller save register, or to spill them into memory. We can do this by extending the def set of `CALL` instructions with every caller-save register.

Callee-save registers can be handled in a similar manner by adding them to the use set of the `RET` instruction. In this case we also need to make sure that their live range is split into shorter segments by copying them into temporaries in the prologue and restoring them back in the epilogue. Otherwise we would lock the register allocator from using any callee-save registers, because physical registers cannot be spilled.

4.4 Linear Scan Register Allocation

Poletto and Sakar proposed [37] a different method of global register allocation called *Linear Scan*. It is not based on graph coloring, but instead does a single pass over the IR while greedily allocating registers to temporaries. This makes linear scan fast, albeit less optimal than graph coloring.

The algorithm requires IR instructions to be numbered in some order (the paper suggests sorting the data-flow graph depth-first). Instead of live ranges used in graph coloring, linear scan works with their conservative approximation – live intervals. A temporary t has live interval $[i, j]$ if there is no instruction with number $j' > j$ such that t is live at j' and there is no instruction $i' < i$ such that t is live at i'.

We can compute live intervals for temporaries from live ranges obtained from liveness analysis with a single pass over the IR. If two temporaries interfere, their live intervals must overlap. We want to assign one of K available registers to each interval so that no overlapping intervals share the same register. If that is not possible, we need to spill some of them into memory.

4.4.1 The Algorithm

The algorithm loops over the live intervals in order of their starting points and keeps track of intervals that overlap at the current program point in the *active* set.

In each iteration, the algorithm first removes intervals that have already ended from *active* and frees their assigned registers. Then, it needs to assign a register to the newly started interval i. This is simple if we have some available, but if there are currently K active intervals, we have to spill one into memory. The implementation in the paper uses a simple heuristic and chooses the interval with the furthest end point. Algorithm 6 contains the three main procedures of linear scan.

4.5 Post Processing

After the register allocation is complete, we still have to do a few more passes over the resulting assembly code.

```
procedure LSRA()
    active ← ∅
    foreach interval i in order of increasing start point do
        ExpireOldIntervals(i)
        if |active| = K then SpillAtInterval(i)
        else
            register[i] ← acquireRegFromPool()
            active ← active ∪ {i}
    end
procedure ExpireOldIntervals(i)
    foreach j ∈ active in order of increasing end point do
        if endpoint(j) ≥ startpoint(i) then return
        active ← active \ {j}
        releaseRegToPool(register[j])
    end
procedure SpillAtInterval(i)
    spill ← last interval from active by endpoint
    if endpoint(spill) > endpoint(i) then
        register[i] ← register[spill]
        location[spill] ← newStackLoc()
        active ← (active \ {spill}) ∪ {i}
    else location[i] ← newStackLoc()
```

Algorithm 6: Pseudocode of the Linear Scan algorithm [37].

Whenever we enter a function, we have to allocate space for the local variables in a stack frame and later deallocate it on function exit. We call the snippets of code responsible for this *prologue* and *epilogue* and they often do some additional actions (for example inserting and checking a stack canary or configuring exception handlers [38]). We could not insert this code during instruction selection or earlier, because the stack frame size could be modified by register spilling.

If we are emitting machine code directly, we also need to transform the generated symbolic instructions into their binary machine code counterparts and replace labels by their real addresses (a process called *assembling*). If the complete program consists of multiple compilation units, we call the linker to resolve symbol cross references and create a single merged executable.

Chapter **5**

Design and Implementation

In this chapter we first discuss the goals of our compiler and how we have integrated it with the existing NIE-GEN toolchain. Then, we explain our choices of methods from the previous chapters for each part of the compiler and describe how we have used them during implementation to reach the stated objectives.

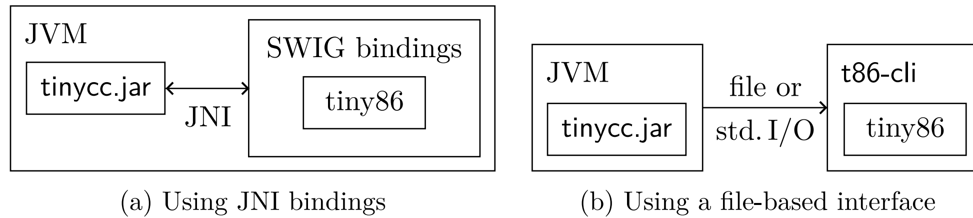
5.1 Design Goals

Instead of building a monolithic program that does the whole compilation in a single pass, we split it into a set of modules and classes with clearly defined interfaces. The modularity allows easily extending the compiler with more frontends and backends in the future and also enables the teacher to reveal only a small part of it to students. They can then implement the rest as their semester projects.

We expect the compiler to be used to process only small programs (in the tens to small hundreds of lines), so we do not have to focus on compilation speed. Instead, we try to implement advanced techniques from the compiler construction course in the most simple and idiomatic way possible, so our implementation can be used as a teaching aid during lectures and tutorials.

Scala supports a vast amount of programming constructs, some of which are not obvious to non-Scala programmers. We want our work to be accessible to the average NIE-GEN student, so we try to keep the use of advanced Scala features to a minimum. Using the simpler Scala 2 instead of Scala 3 gets us halfway there, with the rest being done by avoiding operator overloads, implicits[10] and `for` comprehension except for places where it improves readability (such as in DSLs).

[10] Martin Odersky, the author of Scala, said himself that implicit conversions are evil [39].



(a) Using JNI bindings

(b) Using a file-based interface

Figure 5.1: Two variants of integrating our compiler (tinycc.jar) with tiny86, in (a) tiny86 runs inside the JVM together with the compiler and in (b) the applications are separated on the OS level and communicate using files.

5.2 Extending the NIE-GEN Toolchain

Our compiler should seamlessly integrate with the existing NIE-GEN toolchain, which consists of the tinyC parser and the tiny86 VM, both of which are written in C++ and expose only C++ interface. We have two options how to interact with this interface from Scala (illustrated in fig. 5.1):

1. we can create bindings to use the C++ interface directly from Scala through Java Native Interface (JNI) or a similar bridge, or
2. we have to design a text or binary serialization format and communicate with the toolchain using files or standard I/O.

5.2.1 Calling Native Code from Java

The API surface of the tinyC AST and tiny86 ISA is fairly large (tens of different node and instruction types with a complex class hierarchy), which makes manually creating bindings unfeasible and unmaintainable. In our previous unpublished work we explored methods for automatically creating C++ bindings for Scala using SWIG, which we will now briefly mention.

Simplified Wrapper and Interface Generator (SWIG) is a tool for automatically creating native code bindings for a wide range of language combinations. SWIG can parse a C++ header file with some additional SWIG-specific extensions and generates a set of C++ and Java source files, which together comprise the JNI bindings.

The resulting Java interface is usable from Scala, but there are a few limitations:

- dynamic inheritance does not map cleanly to Java – when a C++ method returns a pointer, it is not resolved to the full type on the Java side and there is no equivalent of `dynamic_cast<T>` (only workarounds), and
- the Java API feels unnatural in Scala – we have to call methods with the `get`, `set`, `is` prefixes and working with Java collections requires using conversions.

Debugging the C++ side is also made difficult, because it runs inside the JVM. Segmentation faults and other exceptions thus crash the entire application. For those reasons we have decided to not use JNI and instead use the file-based approach.

5.2.2 File Input/Output

TinyC is language with a relatively simple syntax, so instead of creating a serializer and deserializer for the AST, we have reimplemented the whole parser from scratch in Scala. Our compiler frontend thus accepts a tinyC source file as an input.

To interface with Tiny86, we have used the text-based assembly format designed by Filip Gregor as part of his interactive debugger [8]. At the time when we started our implementation, the parser (`t86-cli`) was still incomplete, so we have extended it to support all features of tiny86 (notably the data section and floating-point registers). The text-based nature of the format allows the user to examine the assembly code before passing it to the VM. The backend of our compiler thus prints a valid text-based representation of the generated tiny86 program to a file or the standard output.

5.2.3 Improving the Tiny86 VM

During the implementation phase, we have discovered several bugs and shortcomings in the tiny86 VM. We have worked together with the current maintainer Filip Gregor to ensure our fixes reach the upstream.

One of the found and fixed bugs is a data hazard in the CPU pipeline, where the VM crashed during fetch of invalid instruction, which would never be executed (see listing 5.1 for a minimal example). In addition to fixing bugs, we have extended `t86-cli` with the option to report performance statistics of the executed program after the CPU halts.

```
.text
1 CALL 3
2 HALT
3 MOV R0, -1
4 RET
5 MOV R1, [R0]
```

Listing 5.1: The VM crashed as a result of loading from invalid memory address -1 after it fetched the `MOV` instruction on address 5 even though it was never going to be executed.

5.3 Parsing and Typechecking TinyC

In section 5.2 we have decided to implement the tinyC parser from scratch instead of using the one included in the NIE-GEN toolchain (a manually constructed top-down recursive descent parser). For implementing our own parser, we have three feasible options:

1. manually write a recursive descent parser (similar to the one included in the NIE-GEN toolchain),
2. select and use an external parser generator to generate Java or Scala code from the tinyC grammar,
3. implement the parser natively in Scala using parser combinators.

The functional aspect and extensibility of Scala lends itself to the parser combinator approach. Scala makes it easy to implement an internal DSL using overloaded operators, so the parser code can be easily extensible and look similarly to the original parser expression grammar. Another advantage of parser combinators is that we do not need to complicate our build pipeline with external tools and dependencies.

In the past, Scala was distributed with a bundled parser combinator library, which was later separated and is now community-maintained as the scala-parser-combinators package. It provides a set of parser combinators implemented in plain Scala (without macros or any code generation) and related classes for lexing and parsing arbitrary input. In addition, it contains premade lexer and a set of tokens suitable for languages similar to Java. Its main weakness for our purposes is inflexible and unstructured error reporting [40].

5.3.1 A Tiny Parser Combinator Library

We have designed our own tiny parser combinator library with a syntax similar to scala-parser-combinators, but with improved error reporting using the methods we describe in sections 2.1.5 and 2.1.6. While it may seem like a fair bit of upfront work, we could reuse this library for parsing the textual form of our IR and even CLI arguments in addition to tinyC.

For error reporting, we have combined Ford's heuristics [15] with the expectation tree technique designed by Scheidecker [16]. Our parser keeps track of names of expected symbols at the furthest visited position in the input in an expectation tree. If both children of the prioritized choice operator fail at the same position, it creates a new node with expectations of both subexpressions. We can use the p `describedBy` *name* combinator to give a custom name to an expression, which refines the error message with additional context. For simplicity, we have not implemented error recovery, so the parser returns only the first error is encounters. Listing 5.2 shows a sample error message produced by our tinyC parser, which is caused by an incomplete expression.

```
error.tc:2:16: error: while parsing expression, expected '+',
   ↪ '-', '!', '~', '++', '--', '*', '&', integer literal,
   ↪ double literal, char literal, string literal, identifier,
   ↪  '(', 'cast' or 'scan', but got unexpected ';'
    2 |     int i = 1 +;
      |               ^
```

Listing 5.2: An example of an automatically generated parser error message caused by an incomplete expression.

```
object ExprParser extends Parsers {
  lazy val S = E ~ EOI
  lazy val E = T ~ rep('+' ~ T)
  lazy val T = F ~ rep('*' ~ F)
  lazy val F = '(' ~ E ~ ')' | 'a'
}
```

Listing 5.3: A simple combinator parser for PEG from fig. 2.7.

PEG	Scala	description
ε	`success(())`	empty string
a	a	a terminal ($a \in \Sigma$)
A	A	a nonterminal ($A \in N$)
$e_1 e_2$	e_1 `~` e_2	a sequence
e_1/e_2	e_1 `|` e_2	prioritized choice
e^*	`rep(`e`)`	zero or more repetitions
$!e$	`not(`e`)`	a not-predicate

Table 5.1: Mapping between PEG expressions and our combinator library.

A simple parser that just recognizes a PEG can be implemented similarly to listing 5.3. In table 5.1 we show how PEG operators map to our internal Scala DSL. The result of successfully matching some parser expression is the raw parse tree, which is not that useful.

The real power lies in combinators, which manipulate the returned value. The most versatile of them is the map combinator (p `^^` f), which transforms a result of an inner parser p using an arbitrary pure function f. Using this primitive, we can design the parser to directly construct tokens, AST nodes or do any other action that does not have side effects (but we can easily work around this limitation, see section 5.5.8).

5.3.2 Lexical and Syntax Analysis of TinyC

Although a single PEG could express the syntax of tinyC down to the character level, we have decided to use the more standard approach with a separate lexer and parser, both implemented using parser combinators. Our tinyC frontend thus contains two PEGs.

The start expression of the lexical PEG matches a single longest possible token. This is in contrary to the parser, where the start expression succeeds only for a complete program. During its operation, the lexer gradually processes the whole input while passing the matched tokens to the syntax analyzer. We use the following token types for tinyC (here as Scala classes):

```
sealed trait Token
object Token {
  /** An operator, keyword or some other special character. */
  case class Special(value: Symbol) extends Token
  case class Identifier(value: Symbol) extends Token
  case class IntLiteral(value: Long) extends Token
  case class DoubleLiteral(value: Double) extends Token
  case class StringLiteral(value: String, quote: Char) extends
     ↪ Token
}
```

The main tinyC PEG then views those tokens as terminal symbols and uses them to build an AST. We have used the reference tinyC grammar [3] and the AST hierarchy from the parser included in the NIE-GEN toolchain [41].

The grammar of tinyC is ambiguous in whether it should treat an identifier as a named type, because the user can define their own type aliases using `typedef` (same as in regular C) and named structs (tinyC skips the `struct` keyword when referencing structs). We solve this ambiguity by storing the list of currently defined types as an attribute of the input and defining combinators `isNamedType` and `declareNamedType` to query and manipulate it. We cannot use a global state for the same reason that parser expressions cannot have side effects – backtracking and look-ahead using the not-predicate.

The syntax and semantics of tinyC are very close to a subset of C99, in fact the only differences are the omitted `struct` keyword except in declarations, the `cast<T>(e)` operator and in our case 64-bit wide default `int` type. We can easily transpile tinyC source code to C on the AST level, which has allowed us to verify correctness of our tests.

5.4 Semantic and Type Analysis

After parsing, our compiler does two recursive passes over the AST. The first pass builds a map from identifiers to their declarations and the second one is

dedicated to typechecking and returns a map from AST nodes to their types.

TinyC has lexical scoping just like C99, so we keep track of declarations in a lexical stack. Functions, structures and global variables can have multiple forward declarations in addition to their definition. We extend the returned map with the ability to use declarations as a key and use it to store a linked list of previous declarations.

The semantic analyzer also prevents duplicate field names in a `struct`, duplicate cases in a `switch` statement and disallows assignment to a function identifier. Performing those checks earlier simplifies the main IR code generator.

The typechecker is also fairly straightforward, because the type of every AST node in tinyC can be synthesized from types of its children during a single depth-first pass over the tree. Our compiler supports all of the types defined by the official tinyC grammar:

- `void`,
- `char` – 8-bit signed integer in $[-128, 127]$,
- `int` – 64-bit signed integer in $\left[-2^{63}, 2^{63} - 1\right]$,
- `double` – 64-bit IEEE 754 double-precision float,
- a function,
- a structure holding a list of fields, which can be other structures (but recursive declaration is possible only through a pointer),
- a pointer to any of the above, including nested pointers, and
- a statically-sized array – only as a type of a local or global variable.

We automatically promote arithmetic types (`char`, `int`, `double`) in this order when they are used as arguments of a binary arithmetic operator. Our frontend supports implicit and explicit (using `cast<T>(e)`) casting of expression types using C99 rules. Arithmetic types are implicitly compatible with each other, any pointer can be assigned to `void *`, but the opposite direction is possible only through the use of an explicit cast. Likewise, it is possible to explicitly convert between pointer and `int`.

Both semantic and type analyses attempt to continue in case of error. If the typechecker cannot determine type of an expression, it assumes the `void` type. If any of the two analyses detected errors, the compiler does not continue any further and reports them to the user.

In the following section we describe our shared intermediate representation and how we generate it from the tinyC source code.

5.5 Intermediate Representation

From the options presented in section 2.4, we have opted for a flat IR inspired by LLVM. Each instruction in our IR at the same time identifies a virtual SSA register. Emitting SSA code from the frontend is harder and for some C

constructs impossible, so our IR also supports explicit memory load and store operations through specialized instructions.

The SSA property gives our IR all of the advantages of a tree-based IR, because we can easily reconstruct expressions by looking at the implicit data-flow graph. Moreover, explicitly sorting instructions into basic blocks unlocks the potential of doing many lower-level optimizations directly on the IR instead of in target-specific passes.

5.5.1 Structure of an IR Program

A whole IR program consists of a list of functions, each of which contains a list of basic blocks. Every basic block ends with a special terminator instruction, which can only ever exist as the last instruction of a basic block and whose operands specify the next block to be executed. The first executed function on startup is the one named "entry". The first basic block of each function is called the entry block and cannot have any predecessors.

Every instruction has an opcode, name, result type, operands and belongs to a basic block. The instruction name must be unique within a function (with the exception of AllocG, see section 5.5.3) and also identifies the corresponding virtual register.

Each virtual register (instruction) has an associated type, which can be one of void, i64 and double. The IR grammar also accepts a ptr type, but that is just an alias for int64. Pointers in our IR are thus *opaque*, which simplifies the type system, but leaves potential pointer-based optimizations with less information. Operands of instructions are strongly typed and the only way to cast between i64 and double is through the Cast and Bitcast instructions.

The next few sections contain a categorized overview of all supported IR instructions together with a description of their semantics.

5.5.2 Basic Instructions

IImm, FImm (*value* → i64/double) Unlike LLVM, to simplify the object hierarchy, we represent immediate values as regular instructions. The result of IImm is a 64-bit integer constant, the result of FImm is a real constant with double precision.

BinaryArith (*left*: i64/double, *right*: i64/double → i64/double) The IR supports all binary arithmetic operations provided by tiny86. The opcode of each of those instructions starts with one of [IUSF] depending on the expected type of operands. I stands for any integer (signed/unsigned), S for a singed integer, U for an unsigned integer and F for double. Both operands must have the same type, which is also the result type of the instruction.

Cmp (*left*: i64/ptr/double, *right*: i64/ptr/double → i64) Instructions from the Cmp family compare two values with the same type. They use the same letter prefixes as BinaryArith instructions to indicate the accepted type. The result of the comparison is either 0 or 1.

5.5.3 Memory Instructions

The IR contains explicit memory instructions for working with variables on the stack and in the data segment of the program. In addition to the standard types, a variable type can be also a struct or a statically-sized array. We use those types to abstract pointer arithmetic and memory alignment from the frontend and move it into the backend, where the instruction selector can tailor it to the target ISA.

AllocL (*varTy* → ptr) AllocL allocates a variable of the stack and returns its address. The returned pointer is valid until the end of the function.

AllocG (*varTy*, *initData* → ptr) Similarly to AllocL, AllocG allocates a variable in the data segment. In addition to the type, the user can provide an array of 64-bit words, which will be used to initialize the allocated memory at program startup. The name of AllocG must be unique in the entire IR program and it is the only instruction that can be referenced across functions.

Load (*valueTy*, *ptr*: ptr → ptr) The Load instruction retrieves a value of a given type (either i64 or double) from the address specified by *ptr*.

Store (*ptr*: ptr, *value*: i64/ptr/double → void) The Store instruction writes a value to the memory at the address specified by *ptr*.

GetElementPtr (*ptr*: ptr, *index*: i64, *elemTy*, *fieldIndex* → ptr) The GetElementPtr instruction is used to compute offset of an array element or a struct field. It is a simplified version of the equivalent LLVM instruction. In terms of the C language, it computes `&(((`*elemTy* `*)`*ptr*`)[`*index*`].`*fieldIndex*`)`. If *elemTy* is not a struct type, *fieldIndex* must be set to 0.

SizeOf (*varTy* → i64) The SizeOf instruction returns the size of a given variable type in 64-bit words.

5.5.4 Function-Related Instructions

GetFunPtr (*targetFun* → ptr) The GetFunPtr instruction returns address of *targetFun*, which can be indirectly called by CallPtr.

LoadArg ($index \to$ i64/ptr) The LoadArg instruction retrieves value of the *index*-th argument of the parent function. The result is the actual value of the argument, not its address, because depending on calling conventions, it could be present only in a register.

Call (*targetFun*, *args*: i64/ptr/double/void ... → i64/ptr/double/void) The Call calls a function specified by a direct reference. The arguments and result types are dependent on the called function.

CallPtr (*funSig*, *ptr*:ptr, *args*: i64/ptr/double/void ... → i64/ptr/double/void) In addition to direct calls, the IR supports calling function indirectly through their address using CallPtr. The return type and types of arguments are specified by *funSig* and must match with *args*.

5.5.5 Standard Input/Output

The tiny86 VM supports interacting with the host system through the `GETCHAR`, `PUTCHAR` and `PUTNUM` instructions. `GETCHAR` and `PUTCHAR` read and write a single character from the standard input and output, `PUTNUM` prints an integer terminated by a newline (this instruction was added by Filip Gregor to help with testing t86-cli and we have decided to keep it for the same reason). We expose those instructions as the following IR counterparts:

GetChar (() → i64) The GetChar instruction reads and returns a single character from the standard input as an integer in $[0, 255)$. It returns -1 on end of file.

PutChar (*arg*: i64 → void) The PutChar instruction writes a single character ($[0, 255)$) to the standard output. Calling GetChar with an out of bounds argument is an undefined behavior in tiny86.

PutNum (*arg*: i64 → void) The PutNum instruction prints a decimal representation of *arg* terminated by a newline to the standard output.

5.5.6 Special instructions

Phi (*predBlocks*, *args*: i64/ptr/double ... → i64/ptr/double) Phi is a virtual instruction, which implements the SSA ϕ-node used to join values coming from multiple predecessor blocks (see section 2.4). If the previously executed block is the i-th element of *predBlocks*, the instruction evaluates to i-th element of *args*. The size of *predBlocks* and *args* must equal the number of predecessors of the parent basic block and all *args* must have the same type.

Cast (*arg*: i64/double → double/i64) A cast instruction is used to convert between i64 and double types. The IR contains two types of casts: BitcastInt64ToDouble and BitcastDoubleToInt64 reinterpret the binary representation of the argument as the target type, and SInt64ToDouble and DoubleToSInt64 perform a regular conversion between a real number and a signed integer.

5.5.7 Terminator Instructions

We use terminator instructions to direct the control flow at the basic block level. A basic block must have exactly one terminator as its last instruction, which defines its successor blocks.

Ret (*arg*: i64/double/ptr → void) The Ret instruction signalizes the ending of the parent function and returns *arg* to the caller. The type of *arg* must match the return type of the parent function.

RetVoid (() → void) The RetVoid returns execution back to the caller from a function with a void return type.

Halt (() → void) The Halt instruction immediately terminates execution of the program.

Br (*succBlock* → void) The Br instruction represents an unconditional jump. It transfers the execution to the specified successor block.

CondBr (*arg*: i64, *trueBlock*, *falseBlock* → void) The CondBr instruction represents a conditional jump. If *arg* is non-zero, the branch is taken and *trueBlock* is the next to be executed. Otherwise, the execution is transferred to *falseBlock*. We can combine CondBr with the Cmp instruction to build conditions with any of the supported relational operators.

5.5.8 A Textual Representation of the Intermediate Code

During the development of our compiler we needed a way to view inputs and outputs of the individual compiler passes, which led us to design a textual format for our intermediate representation.

We have taken this one step further and designed and implemented a parser, which can read back the text-based format and pass it to the optimizer and target code generation modules of our compiler. This greatly extends the interoperability of our compiler with external tools. The community can develop their own frontends, optimization passes and backends in languages other than Scala and easily integrate them together with the rest of our compiler.

For the parser, we reuse the combinator library designed for frontend, but with a different set of parsing expressions. We still use a separate lexer and parser, but skip the AST this time, because the source code is a direct equivalent to the internal IR program. Because the IR source code can contain forward references and our IR builder is mutable, we cannot directly construct the IR program from within the parser actions. Instead, the parser progressively builds a *function*, which we then call to construct the IR after the whole source code has been successfully parsed.

Listing 5.4 shows the syntax of the textual form of our IR on a sample program. We include the complete PEG grammar in appendix C.

```
fn void entry() {
  entry:
    %str_Hellowor = allocg i64[15], 72 101 108 108 111 44 32 119
       ↪ 111 114 108 100 33 10 0
    %2 = br label %whileCond
  whileCond:
    %1 = phi [ %str_Hellowor, label %entry ],
             [ %7, label %whileBody ]
    %4 = load i64 %1
    %5 = condbr %4, label %whileBody, label %whileCont
  whileBody:
    %ione = iimm 1
    %7 = getelementptr i64, ptr %1, [%ione].0
    %9 = load i64 %1
    %10 = putchar %9
    %11 = br label %whileCond
  whileCont:
    %12 = halt
}
```

Listing 5.4: The IR of the “Hello, world!” program from listing 1.1.

5.6 Compiling TinyC to IR

Our IR is low-level and general enough to be able to represent all tinyC features, but for many constructs the translation is not direct and the frontend has to perform lowering. The compiler visits the AST in post order and emits IR code for each encountered expression.

The following two sections cover interesting problems we have encountered during implementation, but for other specifics we refer the reader to the source code.

```
struct P {
  int x;
  int y;
};
P set_x(int x, P obj) {
  obj.x = x;
  return obj;
}
```

```
ptr set_x(ptr, i64, ptr) {
  entry:
    %retval = loadarg 0
    %x = loadarg 1
    %obj = loadarg 2
    copy obj into retval and
      ↪ set retval->x = x
    %0 = ret %retval
}
```

Figure 5.2: Compilation of a tinyC function which accepts and returns a `struct`. The structures are passed by reference and copied by the callee.

5.6.1 Compiling Structures

The tinyC language fully supports the pass-by-value `struct` type, which means every `struct` variable and argument is independent from each other and assigning to them means performing a deep copy.

The memory instructions in our IR support only primitive types, so we have to emit the copying code in the frontend. Another complication is when a functions has a `struct` as its return value. We have to be careful about who owns the memory and when it may be overwritten.

In our compiler, the caller first allocates memory for the return value and passes its address to the called function as the first argument. The callee is then responsible for copying both the arguments and the return value. The callee then for simplicity returns the pointer to the reserved memory. We copy small structures recursively using a combination of Load, Store and GetElementPtr, but for large structures we call `memcpy(dest, src, n)`, which must be provided by the user. We show a simple example in fig. 5.2.

5.6.2 Compiling Boolean Expressions

To make tinyC compatible with C99, we have decided to implement short circuiting for `||` and `&&` logical operators. A simple way to avoid converting the individual subexpressions into values and storing them in temporary registers or variables is by using a method similar to continuation passing. We define a recursive procedure CompileLogicalExpr(), which takes the expression e, the target block b and two blocks, which should be entered if the condition is evaluated to true (for example the `if` body block) or false (the `if` else block). We show its definition in alg. 7.

```
procedure CompileLogicalExpr(e, b, trueBlock, falseBlock)
    if e is e1 && e2 then
        leftTrue ← newBasicBlock()
        CompileLogicalExpr(e1, b, leftTrue, falseBlock)
        CompileLogicalExpr(e2, leftTrue, trueBlock, falseBlock)
    else if e is e1 || e2 then
        leftFalse ← newBasicBlock()
        CompileLogicalExpr(e1, b, trueBlock, leftFalse)
        CompileLogicalExpr(e2, leftFalse, trueBlock, falseBlock)
    else if e is !e1 then
        CompileLogicalExpr(e1, b, falseBlock, trueBlock)
    else
        reg ← CompileExpr(e, b)
        append CondBr(reg, trueBlock, falseBlock) into b
    end
```

Algorithm 7: A recursive procedure to compile boolean expressions with short-circuiting logical operators.

5.7 The Optimizer

Although we focus this thesis more on the backend part of the compiler, to get the most of our instruction selection we need to perform a few optimizations in the middleend. Most of tinyC statements contain one or more references to a local variable, which our frontend compiles directly to IR as Load and Store instructions. This means that the DFG is split into many small components and not many registers are live at the same time (and almost none across basic blocks).

5.7.1 SSA Construction

Because we want to take advantage of global register allocation, we have to eliminate some of the local variables using SSA conversion. In section 2.6 we described a simple algorithm designed by Braun et. al. for online SSA construction. We have modified this algorithm to work offline in a separate pass over the complete IR program.

For simplicity, we run the SSA construction for each local variable separately. We include AllocL nodes that are used only by Load and as destination of Store instructions, because otherwise there could be indirect reads and modifications of their value. We replace each Load with the result of ReadVariable and call WriteVariable for each Store (see alg. 1). Because we have the full IR CFG available, we could seal a block before filling it if all its predecessors have already been filled, but we have opted for a simpler, but also valid approach of sealing all the blocks at the end.

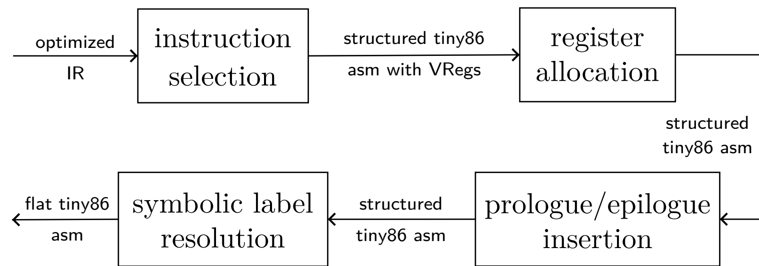


Figure 5.3: Diagram of the backend architecture.

5.7.2 Other Optimizations

Except for SSA construction, our optimizer also contains three very simple additional passes.

Unreachable code elimination does a DFS pass of basic blocks in each function starting from the entry block and removes all non-visited blocks. It also sorts the basic blocks by the time of visit, which ensures that blocks are close to their successors. Because the instruction selector respects the order of IR basic blocks when emitting code, it eliminates many jump instructions.

Another pass shuffles AllocL and AllocG instructions so they are first in the entry block of the parent function and moves global variables to the entry function. This ensures that the instruction selector visits a variable definition before it can see a reference to it.

The last pass ensures that every function has only a single block containing a Ret or RetVoid instruction to prevent duplication of function epilogue. If the function returns a value, it merges them using a ϕ-node.

After the IR has been optimized, it is passed to the compiler backend.

5.8 Overview of the Backend

The backend follows the overall modular architecture of our compiler and is divided into four independent transformation passes as illustrated in fig. 5.3.

First, we take the IR optimized by the middleend and let the instruction selector generate tiny86 assembly with an unlimited amount of temporaries, grouped into basic blocks and functions. The register allocator then maps those temporaries to the physical tiny86 registers, optionally generating code to spill some temporaries onto the stack. The next step completes the assembly by inserting prologue and epilogue and the last step flattens the assembly into a contiguous listing and resolves symbolic labels into addresses.

The output of our backend is an assembly file, which can be directly executed by t86-cli. In the following sections we will describe the modules in order, starting with instruction selection.

5.9 Instruction Selection

In chapter 3 we have described two techniques for instruction selection – macro expansion and tree covering. Macro expansion generates code for each IR instruction individually, whereas tree covering is more advanced and covers the IR program with tiles that can match multiple instructions. Our IR is comparably low level to the tiny86 ISA – there are some instructions that must be implemented by multiple tiny86 instructions (for example Call or CondBr), but for others it is opposite (for example most tiny86 instructions can accept an immediate value as one of their operands).

Because our IR is SSA, the instructions represent nodes in a data-flow graph. We have decided cover this graph with tree tiles by adapting the simple maximal munch algorithm from section 3.2.1. The tiny86 ISA contains a fairly large variety of instructions with a number of them supporting multiple addressing modes. To fully utilize this ISA we need to define more than a hundred of tree tiles, which is infeasible to do manually.

5.9.1 Tree Pattern Matching

We have designed a pattern matching library similar to parser combinators, which we use to define our set of tree tiles. Every tree pattern is a function which accepts a root instruction and returns a list of all matches – it performs recursive matching with full backtracking. Each match provides a value, a list of covered instructions, a list of required instructions (leaves) and a cost (used to penalize some patterns, for example conversion between register types). Because the patterns are regular functions, we can compose them using the choice combinator. Likewise we can alter the value (which is originally the matched instruction and arguments) with an arbitrary pure function using the map combinator.

We use this library to implement a set of tree patterns for generating tiny86 instructions. The tree grammar uses only two nonterminals, Reg and FReg, because otherwise the greedy maximal munch algorithm could get stuck after choosing a tile with a nonterminal that it could not later expand. We use the cost to penalize matching Reg with double and FReg with i64. Because matching the patterns cannot have side effects, we use the same workaround as when parsing our textual IR (section 5.5.8): the value of the pattern matches is a function, which generates the code using a context passed as an argument and returns the register number holding the result.

Listing 5.5 contains an example implementation of one such rewrite rule. On the left side is the Reg nonterminal and the right side matches an IAdd instruction with two operands stored in integer registers. Note that `left` and `right` are not register numbers, but functions that return it after they are called with the context.

```
GenRule(RegVar, Pat(IAdd, RegVar, RegVar).map({
  case (insn, left, right) => (ctx: Context) => {
    val leftReg = left(ctx)
    val rightReg = right(ctx)
    val resReg = ctx.freshReg()
    ctx.emit(MOV, resReg, leftReg)
    ctx.emit(ADD, resReg, rightReg)
    resReg
  }
}))
```

Listing 5.5: An example Scala implementation of a rewrite rule for the IAdd instruction.

5.9.2 Maximal Munch

For pattern selection we use the greedy maximal munch algorithm. We start at the roots (nodes with zero in-degree) of the data-flow graph and greedily select patterns in depth-first order. Because our IR is a DAG, it can contain shared nodes and the algorithm has to handle the case when a node has already been covered.

If it has been covered as a root node, we reuse its result – we perform edge splitting. If it has been covered as an inner node, we cover the node multiple times – this is equal to node duplication mentioned in section 3.2.3. We constructed our pattern set so that instructions with side effects are only matched as roots, so those are never duplicated.

The result of the maximal munch algorithm is a map from IR instructions to matched rewrite rules, where every instruction is covered by at least one pattern. We then do two additional passes over the function body to collect and emit assembly instructions for the selected tiles. Before emitting code for a basic block terminator, we emit `MOV` instructions required by ϕ-node elimination.

5.9.3 Calling Conventions

Emitting code for function calls requires specifying calling conventions. For simplicity, we decided to store function arguments on the stack and the return value in the first integer register `R0`. Before emitting `CALL`, the caller pushes the arguments onto the stack in reverse order. The caller then cleans the stack again after the called function returns. All integer registers are callee-save and all float registers are caller-save.

We show the stack layout in fig. 5.4. Because memory in tiny86 is addressed in 64-bit words which fit a value of every IR type (i64, double, ptr), we do not

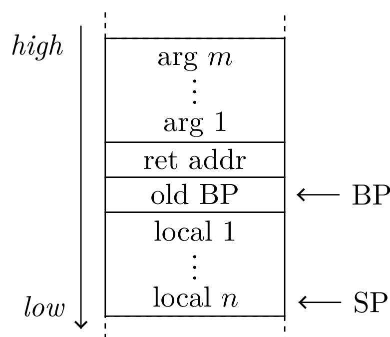


Figure 5.4: Stack layout after entering a function. The first argument is at `[BP + 2]`, the first local variable is at `[BP + -1]`.

```
FADD VF7, 1.0
```

(a)

```
MOV VR9, [BP + -1]
MOV F0, VR9
FADD F0, 1.0
MOV [BP + -1], F0
```

(b)

```
MOV R0, [BP + -1]
MOV F0, R0
FADD F0, 1.0
MOV [BP + -1], F0
```

(c)

Figure 5.5: Spilling `VF7` requires a new integer temporary `VR9`, which the integer register allocator later assigns to `R0`.

have to deal with stack alignment. The tiny86 stack starts at the highest memory address and grows towards zero.

5.10 Register Allocation

The second big component of the backend that comes after the instruction selector is the register allocator. Tiny86 has a configurable amount of two kinds of registers – integer (`Rx`) and floating point (`Fx`). Float registers are accepted only by a handful of float instructions and cannot be used for other operations and this holds vice versa for integer registers.

We solve this additional complexity by first allocating all float registers and then running the entire register allocator again, this time only on integer registers. We do it in this order, because a spilled float register can be loaded only through an intermediate integer register. We show an example of spilling in fig. 5.5.

To easily debug the instruction selector, we have designed a stand-in register allocator that relies on the ability to configure tiny86 with an arbitrary register count. It works by directly mapping the virtual registers to the available range of physical registers and can be used only for small programs. It treats all registers as callee-save (including float registers), because only then it knows the specific subset of used registers to back up and restore.

For regular operation, we have chosen the graph coloring register allocation algorithm with move coalescing designed by Appel [19]. It works more optimally than linear scan and its main drawback, slower compilation speed, is not an issue for us. We design our compiler for educational purposes and register allocation by graph coloring is one of the main topics covered in the NIE-GEN course.

Another advantage of graph coloring register allocation is that with a simple modification of the def and use sets of call-related instructions it can automatically save and restore caller and callee-save registers. We use the following rules:

- def(`CALL Rx`) := {caller-save registers, return value register}, and
- use(`RET Rx`) := {callee-save registers, return value register}.

Additionally, we backup all callee-save registers to fresh temporaries in function prologue and restore them in function epilogue. This is important because otherwise the register allocator would not have any temporaries to spill (it cannot spill precolored physical registers). If some of the caller-save registers stay unused, the register allocator coalesces the temporaries and eliminates the inserted move instructions.

5.10.1 Liveness Analysis

Before we can build and color the interference graph, we need to determine live ranges of each register using liveness analysis. We compute set of live-out temporaries for each basic block using the monotone framework introduced in section 4.2. Appel's algorithm then uses these sets to determine live variables before and after each instruction and to construct the interference graph.

Before we can solve the dataflow equations, we need to compute defines and uses for each basic block using alg. 8. The algorithm starts with empty sets and updates them while looping backwards over the body of a basic block.

procedure GetBlockDefUse(b)
    $bbdef \leftarrow \emptyset$; $bbuse \leftarrow \emptyset$
    foreach $i \in$ instructions in b in reverse order **do**
        $bbdef \leftarrow bbdef \cup \text{def}(i)$
        $bbuse \leftarrow (bbuse \setminus \text{def}(i)) \cup \text{use}(i)$
    end
    return $(bbdef, bbuse)$

Algorithm 8: Computation of def and use sets for a whole basic block.

To find the least fixed point of the dataflow equations, we use the naive solver from alg. 5.

5.10.2 Graph Coloring

We use the rest of Appel's register allocation algorithm without any functional changes. The algorithm builds the interference graph using the information obtained from the liveness analysis and then alternates between simplification and move coalescing until the interference graph is empty. Then, it tries to rebuild the graph while assigning colors and if that fails, marks nodes for spilling.

To decide what node to spill we use the heuristic suggested by Appel [19]. We spill the candidate with the lowest cost computed from the count of its references inside and outside of a cycle in the CFG as follows:

$$spillCost(t) = \begin{cases} +\infty & t \text{ is precolored or was created by spilling another temporary,} \\ inLoop(t) \cdot 10 + outLoop(t) & \text{otherwise.} \end{cases}$$

The purpose of this heuristic is to avoid spilling registers used by instructions that could be executed frequently. To determine whether a basic block is part of a cycle, we use Kosaraju's algorithm to find strongly connected components of the CFG. A node is part of a cycle if its component has size greater than one or it is a self-loop.

After we successfully find a valid coloring for the interference graph, we remove redundant `MOV Rx, Rx` instructions.

5.11 Post Processing

After we have completed the register allocation, we know the size of local variables and we can insert a function prologue and epilogue by replacing the corresponding markers inserted during instruction selection. In the prologue we push the old base pointer to the stack, start a new call frame and allocate space for local variables. In the epilogue we restore the old call frame and return execution to the caller. Listing 5.6 contains the exact inserted tiny86 assembly code.

```
PUSH BP
MOV BP, SP
SUB SP, −localsSize
... function body ...
MOV SP, BP
POP BP
RET
```

Listing 5.6: Tiny86 listing of the inserted function prologue and epilogue.

As the last step before printing the generated tiny86 assembly we flatten the functions and basic blocks into a single continuous listing and resolve symbolic labels to their numeric addresses.

5.12 Implementation

In this section we describe how we have tackled the implementation of our compiler in Scala 2. As we stated in section 5.1, we have striven to keep the our code as readable as possible, so it is can be understood and used by NIE-GEN students. The key means to do this has been separating the compilation process into small coherent stages (compiler passes), that do only a single operation. The main application then calls those passes in a specific order to progressively translate the tinyC source code to tinyC assembly. Our implementation follows the don't repeat yourself (DRY) principle by moving the commonly used code into their own functions and classes.

We have implemented everything in a single sbt project named tinycc. We import only one external dependency scalatest which we use for unit and integration testing. For better clarity, we have split the project into 5 packages:

- backend – everything related to tiny86, instruction selection and register allocation,
- cli – the command line user interface,
- common – the intermediate representation and optimizer,
- frontend.tinyc – the tinyC frontend (parser, typechecker, tinyC to IR compiler), and
- util – the parser combinator library, logger and other miscellaneous classes used by other packages.

5.12.1 The Parser Combinator Library

We have implemented our parser combinator library in the util.parsing.combinator package and it is used by both the frontend, the middleend and the command line interface of our compiler.

A `Parser[T]` extends `(Input => Result[T])`, where `Input` is a subclass of `Reader`. The reader represents an immutable sequence of terminals, but keeps track of the current position in the original input (line, column and offset). A result can be either `Accept[T]` if the parser has successfully matched a prefix of the input or `Reject`, which signalizes a rejection with an information about what was expected. A rejection can be optionally fatal, which means that no backtracking will be performed. The rest of the `Parsers` trait then contains definitions of parser combinators.

`StringParsers` and `SeqParsers` then provide specialized version of `Reader` and combinators for parsing strings and generic sequences. `Lexical` and

Scanners then build on top of StringParsers and can be used to implement a lexical analyzer after the user provides parsing expressions for tokens and ignored whitespace.

5.12.2 The TinyC Frontend

We use this parser combinator library to implement the tinyC lexer and parser in the frontend.tinyc.parser package. The tinyC lexer exposes its interface as reader of tokens, which is accepted by the parser. The TinyCParser requests tokens from the lexer as needed and at the end returns an instance of the root AST node, AstProgram. Our AST is completely immutable.

The SemanticAnalysis and TypeAnalysis classes take the AstProgram as input and return Map[AstIdentifierOrDecl, IdentifierDecl], respectively Map[AstNode, Ty]. IdentifierDecl is a reference to either variable, function or function argument declaration. Ty then represents a tinyC type.

The TinyCCompiler then takes the AstProgram and the declaration and type maps and constructs an IrProgram using builders provided by the common.ir package.

We have implemented both the analyses and the compiler itself as recursive passes over the AST, which use pattern matching to perform different actions depending on visited node type. Errors are thrown as exceptions.

```
import tinycc.frontend.tinyc._
val scanner = new parser.Lexer.Scanner(sourceCodeStr)
val ast = parser.TinyCParser.parseProgram(scanner)
val decls = new analysis.SemanticAnalysis(ast).result()
val types = new analysis.TypeAnalysis(ast, decls).result()
val irProgram = new TinyCCompiler(ast, decls, types).result()
```

Listing 5.7: Example of how the single-purpose classes can be combined to compile a tinyC source code into IR. We also provide a convenience function to do these steps at once.

5.12.3 Intermediate Representation

During implementation of the intermediate representation we had to find a balance between mutability and immutability of its components. Immutable code and functional programming can be less prone to errors caused by side effects and it is the typical way to do things in Scala. On the other hand, our compiler frequently needs to manipulate the IR by inserting, removing and moving instructions and basic blocks around. With an immutable IR it would mean constantly recreating its parts.

In the end we have settled on mostly mutable IR, which we have implemented in the common.ir package. The optimizer often needs access to a list of uses of a particular instruction, so we track them using a mutable smart

reference implemented as `Ref[T]`. Whenever a reference is set to a particular object (`T <: UseTracking[_, _]`), it informs the object about itself and the same thing happens when the reference is cleared.

We have provided helpers to move instructions between basic blocks in the `IrManipulation` class and builders to quickly construct and fill a basic block, function or a whole IR program in the `IrBuilder` file, which are used by the frontend and the optimizer.

5.12.4 The Optimizer

The common package also contains implementation of the optimizations. Each optimization pass extends `ProgramTransform[IrProgram]`, which is a simple interface with a single method `transform(program: IrProgram): Unit`, which performs the optimization by directly mutating the provided IR program. We have implemented the following optimizations, executed in this order:

1. `BasicBlockScheduling` – sorts basic blocks using a simple DFS pass so jumps are often followed by their destination and removes unreachable blocks,

2. `AllocOrdering` – moves all AllocL instructions to the beginning of a function and all AllocG instructions to the beginning of the entry function,

3. `MemToReg` – eliminates some local variables (AllocL instructions) by performing SSA construction, and

4. `SingleFunExit` – ensures that a function contains only a single Ret or RetVoid instructions by creating a new exit block that merges the return values with a Phi.

The `Optimizer` implements the same interface as the classes above and is responsible for running the optimizations in a set order. The `MemToReg` and `SingleFunExit` optimizations can be toggled using a parameter passed to the constructor.

5.12.5 Instruction Selection

After the optimizer transforms the IR, the application hands off the code to the instruction selector. We have implemented it in two parts: the backend.insel package contains the general tree pattern matching library and the maximal munch algorithm, and backend.t86.insel contain the classes specific to tiny86 – the code generator itself and the required set of patterns. We compose both parts together into a complete instruction selector like we show in listing 5.8.

`GenericNaiveRA[T]`	or	`GenericGraphColoringRA[T]`
	+	
`T86RegRegisterAllocator`	or	`T86FRegRegisterAllocator`
	+	

additional code specific to the combination mostly related to spilling

Figure 5.6: Constructing a register allocator from the allocation algorithm, register-specific information and additional spilling-related code.

```
import tinycc.backend.t86
import tinycc.backend.insel
val t86Program = (
  new t86.insel.T86TilingInstructionSelection(irProgram)
    with t86.insel.GenRules
    with insel.MaximalMunch).result()
```

Listing 5.8: Using Scala class composition to construct and run a tiny86 maximal munch instruction selector with the default ruleset.

The result of the instruction selection is an `T86Program`, which is the root of our hierarchical representation of tiny86 assembly. A `T86Program` consists of functions (`T86Fun`), which contain a list of basic blocks (`T86BasicBlock`), which at last contain the tiny86 instructions themselves. Every `T86Program`, `T86Fun` and `T86BasicBlock` contains a reference to the original IR object.

5.12.6 Register Allocation

We use class composition again in our register allocator, this time to share code between allocation of integer and float registers. We have implemented both the naive register allocator which requires an unlimited amount of registers (`GenericNaiveRegisterAllocator[T]`) and the graph coloring allocator (`GenericGraphColoringRegisterAllocator[T]`), where `T` can be either `Operand.Reg` or `Operand.FReg`. We then extend the generic base with one of `T86RegRegisterAllocator` and `T86FRegRegisterAllocator`, which provide methods to get the list of registers defined and used by an instruction or a basic block and other information about the available registers of the given type.

We then compose a complete register allocator for one register type using classes from backend.t86.regalloc via the scheme in fig. 5.6. The backend then calls the float register allocator and integer register allocator in succession.

Graph Coloring Implementation We have implemented the graph coloring register allocator itself in three stages. First, `LivenessAnalysis` computes live-in and out temporaries for each basic block, which is then used by

`InterferenceGraph` to construct a graph of interfering temporaries and finally `InterferenceGraphColoring` implements the graph coloring and move coalescing itself. We use `WorklistSet` to ensure that every node and move instruction is always in exactly one of the worklists defined in section 4.3.3.

5.12.7 Command-Line Interface

The final component of the compiler that is responsible for calling the other modules is the command-line user interface, which resides in the cli package.

We have decided to implement the CLI rather unusually by using our parser combinator library. Instead of parsing characters of a string, we treat whole arguments as terminals.

The CLI (named tinycc) allows the user to run all three parts (frontend, middleend, backend) separately with file I/O (compile-to-ir, optimize and codegen), or perform the entire compilation from start to finish (compile). To monitor the time taken in the individual components of the compiler, we have implemented a simple profiler, which can be enabled using the --profile option.

5.13 Documentation

As we stated in chapter 1, we want other students to use parts of our implementation as foundation of their NIE-GEN compiler projects. For that reason we have documented our implementation using Scaladoc annotations, which are automatically displayed by the IDE. Additionally, we include a brief user guide in appendix B and the CLI contains built-in description of all available commands, which can be displayed by tinycc help.

Chapter 6

Evaluation

The implementation itself is only one phase of the development cycle. In this chapter, we first evaluate the correctness of our tinyC compiler using two types of automated tests. We then use the rest of the chapter to experimentally evaluate code size and execution time of typical code generated by the compiler.

6.1 Unit Tests

We have designed a range of unit tests to verify the correctness of individual classes and components. To manage and run them, we use the popular ScalaTest library. All unit tests are in the `src/test/scala` folder and their names correspond to the tested classes.

One has to write a lot of unit tests to gain any meaningful code coverage, because the tests are evaluating only a small part of the whole program in isolation. We use them primarily to test our parsers and analyses, but the bulk of our compiler is tested only by larger integration and end to end tests.

6.2 Integration and End to End Tests

Testing compilers is a hard problem, because they accept a wide range of inputs (any valid source code). Some compilers compile their own source code as one of the tests, but this is possible only for *self-hosting* compilers, which is not our case. We have designed a suite of 44 small tinyC programs which gradually test the features supported by our compiler. Every test is a self-contained `*.c` file in the `examples/` directory and provides standard input and expected standard output using special comments.

`TinyCFrontendTest` verifies correctness of the parser and frontend by compiling all of the test cases and calling `validate()` on the generated IR pro-

grams. This method statically checks some properties that must hold in a valid program, such as that:

- all operands are set and have the expected result type,
- all basic blocks are terminated, and
- there are no duplicate functions, basic blocks or instructions in the program.

The `End2EndTest` then uses the same test cases as input, but this time compiles them into assembly and executes the generated code on the tiny86 VM. The test then verifies that the executed program printed the correct output.

To verify the correctness of this reference output, we have implemented a simple AST-level transpiler of tinyC into C. The run_tests.sh script transpiles every test case to C, compiles it using GCC and runs it on the host. We use a small runtime to provide the builtin `scan`, `print` and `printnum` functions.

We can execute both the unit and end to end tests with the sbt test command.

6.3 Benchmarks

To determine the impact of the implemented optimizations, instruction selection and register allocation, we evaluate the code size and execution time of two small programs for solving numerical problems. We have intentionally selected computation-heavy problems so we are not limited by memory access latency.

The first program (factorize.c) computes and prints prime factors of an integer. We have replaced division and modulo by 2 with equivalent bitwise operations.

```
void factorize(int n) {
    while(n > 0 && (n & 1) == 0) {
        printnum(2);
        n = n >> 1;
    }
    for(int i = 3; i * i <= n; i = i + 2) {
        while(n % i == 0) {
            printnum(i);
            n = n / i;
        }
    }
    if(n > 1) printnum(n);
}
```

Listing 6.1: factorize.c: A program for computing prime factors of a positive integer.

The second sample program determines whether the Collatz conjecture holds for a given positive integer. Collatz states in his conjecture that for any positive starting integer, if we repeatedly apply the following operation, we will always eventually obtain 1:

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod{2} \\ 3n+1 & \text{if } n \equiv 1 \pmod{2} \end{cases}$$

Whether this holds for any positive integer is one of the most famous unsolved problems in mathematics and there have not been found any counterexamples yet. The program collatz.c computes the number of steps it takes to reach 1 starting at n.

```
void collatz(int n) {
  int i = 0;
  while(n > 1) {
    if((n & 1) != 0) {
      n = 3 * n + 1;
    } else {
      n = n >> 1;
    }
    i++;
  }
  printnum(i);
}
```

Listing 6.2: collatz.c: A program that prints the number of reduction steps required to reach 1 from a positive starting integer n.

6.3.1 Measuring Effects of SSA Construction

We have compiled both of the programs with and without enabled optional optimizations (SSA construction) and in table 6.1 we compare the size of generated assembly code, number of execution ticks of the VM and average number of executed instructions per tick (IPT) on three sample inputs. We use the default number of integer registers (4) and all selected metrics are independent on the host system. Tiny86 contains a logger (enabled by passing -stats to t86-cli) that collects information about executed instructions and provides values for the last two execution metrics.

In both cases enabling SSA construction produced smaller and faster code. Utilizing registers instead of memory decreased the average instruction latency and more instructions could be executed in the same amount of ticks. For the two harder inputs the optimized assembly of collatz.c executed around 75% faster. We include the `collatz` function compiled with optimizations in listing 6.3. The rest of the source code and tiny86 assembly is available in the directory benchmarks/results/ of the electronic attachment.

6.3.2 Measuring Effects of Improved Backend

In this second experiment we compare the compiler presented in this thesis (tinycc) with another, simpler tinyC compiler we created in the past (tinyc). This older version does not support SSA construction, uses a very simple instruction selection and a simple local register allocator similar to the one described in section 4.1. Through this comparison we can measure the improvements in code size and execution speed offered by the tree tiling instruction selection and global graph coloring register allocator implemented in tinycc.

The older compiler does not support the `printnum(n)` construct, so we had to provide an equivalent function as part of the source code. In table 6.2 we compare tinycc with disabled optimizations to the older tinyc. In both cases we have set the number of integer registers to 4. As with the previous experiment, we include the raw files in the electronic attachment.

We can see that the advanced instruction selection and register allocation improve the code size dramatically and in case of collatz.c the code executes 50% faster even with disabled optimizations than when compiled with tinyc, and enabling them would further widen the gap.

We believe that the throughput got lower because due to the reduced code size, jump statements are closer together (there is less `MOV` instructions) and thus the CPU flushes its pipeline more often. We could probably obtain better throughput by improving the branch predictor in tiny86, which is currently very rudimentary [4].

	without -O			with -O			speedup
	size	ticks	IPT	size	ticks	IPT	
factorize(10)		272	0.213		212	0.231	1.28
factorize(64)	70	560	0.218	54	399	0.248	1.40
factorize(68767889)		53098	0.239		30707	0.268	1.73
collatz(31)		8374	0.198		4538	0.287	1.85
collatz(42)	47	742	0.204	42	450	0.278	1.65
collatz(837799)		40866	0.198		21932	0.288	1.86

Table 6.1: Comparison of size and performance of generated code with and without enabled SSA construction. With enabled optimizations, the code is in both cases smaller and the program finishes execution faster.

	tinyc			tinycc without -O			speedup
	size	ticks	IPT	size	ticks	IPT	
factorize(10)		769	0.338		650	0.215	1.18
factorize(64)	261	2054	0.353	120	1694	0.217	1.21
factorize(68767889)		93951	0.410		54400	0.239	1.73
collatz(31)		14968	0.337		8827	0.199	1.70
collatz(42)	172	1404	0.353	93	931	0.206	1.51
collatz(837799)		71598	0.369		41319	0.198	1.73

Table 6.2: Comparison of size and performance of generated code by our compiler (tinycc) with a different, older compiler (tinyc) with only rudimentary instruction selection and a simple local register allocator. tinycc produces smaller and faster code for both programs. The numbers in the second column are different from table 6.1 because here they include the required `printnum` implementation.

```
# collatz$entry:
   4  PUSH BP
   5  MOV BP, SP
   6  SUB SP, 2
   7  MOV [BP + -2], R2
   8  MOV [BP + -1], R3
#     [BP + 2] -> arg #0
   9  MOV R0, [BP + 2]
  10  MOV R2, 0
# collatz$whileCond:
  11  CMP R0, 1
#     26 -> collatz$whileCont
  12  JLE 26
# collatz$whileBody:
  13  MOV R3, R0
  14  AND R3, 1
  15  CMP R3, 0
#     23 -> collatz$ifFalse
  16  JE 23
# collatz$ifTrue:
  17  MOV R3, 3
  18  IMUL R3, R0
  19  INC R3
# collatz$ifCont:
  20  INC R2
  21  MOV R0, R3
#     11 -> collatz$whileCond
  22  JMP 11
# collatz$ifFalse:
  23  RSH R0, 1
  24  MOV R3, R0
#     20 -> collatz$ifCont
  25  JMP 20
# collatz$whileCont:
  26  PUTNUM R2
  27  MOV R2, [BP + -2]
  28  MOV R3, [BP + -1]
  29  MOV SP, BP
  30  POP BP
  31  RET
```

Listing 6.3: A complete listing of `void collatz(int)` from collatz.c compiled by tinycc with optimizations. The tiny86 assembly does not support symbolic labels, so they are represented as comments.

Conclusion

In this thesis we have designed and implemented a modular tinyC compiler written in Scala, that integrates with the tiny86 VM via a text-based assembly format. The compiler is composed of three main separable components (frontend, middleend and backend) and can be easily extended either in Scala by depending on some of its parts or in any other language through our text-based intermediate representation. The simplicity and extensibility of our architecture makes it a great tool for educational purposes in compiler construction courses such as NIE-GEN.

We have described and implemented a range of advanced techniques from compiler construction with focus on the backend that allow the compiler to efficiently utilize more complex instructions and all available registers of tiny86. We have checked the correctness of our work and provided documentation of both the API and the CLI user interface.

We have demonstrated on two examples that the selected algorithms and optimizations perform well and produce smaller code that executes faster than if we compile the same programs with a simpler compiler.

Future Work

Our general purpose IR and parser combinator library allow easy development of other frontends. In the future we could extend the compiler with support for the microC language, which would enable this work to be used in the NIE-APR course.

The middleend currently contains only one significant optimization, SSA construction. Our SSA-based IR simplifies implementation of other static analyses and optimizations, which could be explored in further work. Easy targets are dataflow analyses taught in NIE-APR such as constant propagation or available expression analysis.

The compiler is also open to additions of new backends. Register-based targets such as x86_64 or RISC-V could reuse most of our tiny86 backend

including the global register allocator, while stack-based targets such as the JVM could still make some use of the implemented generic tree covering instruction selection.

Bibliography

[1] Møller, A.; Schwartzbach, M. I. *Static Program Analysis.* Aarhus, Denmark: Department of Computer Science, Aarhus University, October 2018. [viewed date 2 May 2023]. Available from: http://cs.au.dk/~amoeller/spa/

[2] Johnson, S. C. A Tour Through the Portable C Compiler. In: *Unix Programmer's Manual*, chapter 33. Murray Hill, New Jersey, USA: AT&T Bell Laboratories, 7th edition, 1981, ISBN 0-03-061743-X.

[3] Máj, P. TinyC Language Reference. In: *TinyC Project Repository – GitLab* [online]. 2023. [viewed date 2 May 2023]. Available from: https://gitlab.fit.cvut.cz/NI-GEN/ni-gen-23/-/blob/main/LANGUAGE.md

[4] Strejc, I. *Tiny x86 - Architecture Simulator for Educational Purposes.* Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2021. Available from: http://hdl.handle.net/10467/94644

[5] Odersky, M.; Spoon, L.; et al. *Programming in Scala.* ITpro collection, Artima, Incorporated, 2019, ISBN 9780981531618. Available from: https://books.google.com/books?id=ph0yzAEACAAJ

[6] Aho, A.; Lam, M.; et al. *Compilers: Principles, Techniques, & Tools.* Pearson/Addison Wesley, 2007, ISBN 9780321486813. Available from: https://books.google.com/books?id=dIU_AQAAIAAJ

[7] International Standardization Organization. *ISO/IEC 9899:TC3: Programming Language: C.* Technical report, ISO/IEC JTC1/SC22, 2007. Available from: https://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf

[8] Gregor, F. *Tiny86 Debugger.* Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2023.

[9] Lattner, C.; Adve, V. LLVM: a compilation framework for lifelong program analysis & transformation. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86, doi: 10.1109/CGO.2004.1281665.

[10] Král, V. *Ahead-of-time compiler for the microC language.* Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2022. Available from: `http://hdl.handle.net/10467/101065`

[11] Ford, B. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. New York, NY, USA: Association for Computing Machinery, jan 2004, ISSN 0362-1340, p. 111–122, doi:10.1145/982962.964011. Available from: `https://doi.org/10.1145/982962.964011`

[12] Ford, B. Packrat Parsing: Simple, Powerful, Lazy, Linear Time, Functional Pearl. 2002: p. 36–47, doi:10.1145/581478.581483. Available from: `https://doi.org/10.1145/581478.581483`

[13] Hutton, G. Higher-order functions for parsing. *Journal of Functional Programming*, volume 2, no. 3, 1992: p. 323–343, doi: 10.1017/S0956796800000411. Available from: `https://doi.org/10.1017/S0956796800000411`

[14] Maidl, A. M.; Mascarenhas, F.; et al. Exception Handling for Error Reporting in Parsing Expression Grammars. *PROGRAMMING LANGUAGES, SBLP 2013*, volume 8129. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ISBN 0302-9743, pp. 1–15.

[15] Ford, B. *Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking.* Master's thesis, Massachusetts Institute of Technology, 2002. Available from: `http://hdl.handle.net/1721.1/87310`

[16] Scheidecker, A. Parser Combinators and Error Reporting [online]. Dec 2012. [viewed date 2 May 2023]. Available from: `https://www.scheidecker.net/2012/12/03/parser-combinators/`

[17] Křikava, F. The µC Programming Language. In: *Selected Methods for Program Analysis (NI-APR) – FIT CTU Course Pages* [online]. Mar 2023. [viewed date 2 May 2023]. Available from: `https://courses.fit.cvut.cz/NI-APR/microc.html`

[18] Chow, F. Intermediate Representation: The Increasing Significance of Intermediate Representations in Compilers. *Queue*, volume 11, no. 10, Oct 2013: p. 30–37, ISSN 1542-7730, doi:10.1145/2542661.2544374. Available from: `https://doi.org/10.1145/2542661.2544374`

[19] Appel, A.; Palsberg, J. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002, ISBN 9780521820608. Available from: https://books.google.com/books?id=IjKIngEACAAJ

[20] Free Software Foundation. GNU Compiler Collection (GCC) Internals. In: *GCC, the GNU Compiler Collection* [online]. 2023. [viewed date 2 May 2023]. Available from: https://gcc.gnu.org/onlinedocs/gccint/

[21] Bellard, F.; et al. Tiny C Compiler Reference Documentation [online]. Aug 2018. [viewed date 2 May 2023]. Available from: https://bellard.org/tcc/tcc-doc.html

[22] Cytron, R.; Ferrante, J.; et al. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, volume 13, no. 4, Oct 1991: p. 451–490, ISSN 0164-0925, doi:10.1145/115372.115320. Available from: https://doi.org/10.1145/115372.115320

[23] Braun, M.; Buchwald, S.; et al. Simple and Efficient Construction of Static Single Assignment Form. *COMPILER CONSTRUCTION, CC 2013*, volume 7791. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ISBN 0302-9743, pp. 102–122.

[24] Blindell, G. *Instruction Selection: Principles, Methods, and Applications*. Springer International Publishing, 2016, ISBN 9783319340197. Available from: https://books.google.com/books?id=yp9PDAAAQBAJ

[25] Fišer, P.; Schmidt, J. Global Methods. In: *Combinatorial Optimization (NIE-KOP)*. 2022. [viewed date 2 May 2023]. Available from: https://moodle-vyuka.cvut.cz/pluginfile.php/528885/mod_page/content/6/Lecture12-Global.pdf?time=1670775676726

[26] Garey, M.; Johnson, D. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Mathematical Sciences Series, Freeman, 1979, ISBN 9780716710448. Available from: https://books.google.com/books?id=fjxGAQAAIAAJ

[27] Koes, D. R.; Goldstein, S. C. Near-Optimal Instruction Selection on Dags. In: *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, New York, NY, USA: Association for Computing Machinery, 2008, ISBN 9781595939784, p. 45–54, doi:10.1145/1356058.1356065. Available from: https://doi.org/10.1145/1356058.1356065

[28] Fauth, A.; Hommel, G.; et al. Global code selection for directed acyclic graphs. *Compiler Construction*, volume 786. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994;, ISBN 0302-9743, pp. 128–142.

[29] Arnold, M. Matching and Covering with Multiple-Output Patterns. Technical report, Delft University of Technology, Delft, The Netherlands, 1999.

[30] Khedker, U.; Sanyal, A.; et al. *Data Flow Analysis: Theory and Practice*. CRC Press, 2017, ISBN 9780849332517. Available from: `https://books.google.com/books?id=9PyrtgNBdg0C`

[31] Miller, P. L. *Automatic Creation of a Code Generator from a Machine Description*. Master's thesis, Massachusetts Institute of Technology, 1971. Available from: `https://hdl.handle.net/1721.1/149399`

[32] Davidson, J.; Fraser, C. The Design and Application of a Retargetable Peephole Optimizer. *ACM Trans. Program. Lang. Syst.*, volume 2, 04 1980: pp. 191–202, doi:10.1145/357121.357129. Available from: `https://doi.org/10.1145/357121.357129`

[33] Davidson, J. W.; Fraser, C. W. Code Selection through Object Code Optimization. *ACM Trans. Program. Lang. Syst.*, volume 6, no. 4, oct 1984: p. 505–526, ISSN 0164-0925, doi:10.1145/1780.1783. Available from: `https://doi.org/10.1145/1780.1783`

[34] Chaitin, G. J.; Auslander, M. A.; et al. Register allocation via coloring. *Computer Languages*, volume 6, no. 1, 1981: pp. 47–57, ISSN 0096-0551, doi:https://doi.org/10.1016/0096-0551(81)90048-5. Available from: `https://www.sciencedirect.com/science/article/pii/0096055181900485`

[35] Briggs, P.; Cooper, K. D.; et al. Improvements to Graph Coloring Register Allocation. *ACM Trans. Program. Lang. Syst.*, volume 16, no. 3, may 1994: p. 428–455, ISSN 0164-0925, doi:10.1145/177492.177575. Available from: `https://doi.org/10.1145/177492.177575`

[36] George, L.; Appel, A. W. Iterated Register Coalescing. *ACM Trans. Program. Lang. Syst.*, volume 18, no. 3, may 1996: p. 300–324, ISSN 0164-0925, doi:10.1145/229542.229546. Available from: `https://doi.org/10.1145/229542.229546`

[37] Poletto, M.; Sarkar, V. Linear Scan Register Allocation. *ACM Trans. Program. Lang. Syst.*, volume 21, no. 5, sep 1999: p. 895–913, ISSN 0164-0925, doi:10.1145/330249.330250. Available from: `https://doi.org/10.1145/330249.330250`

[38] Zahradnický, T.; Kokeš, J. Introduction to Reverse Engineering, Stack Frame Analysis. In: *Reverse Engineering (NIE-REV)* [online]. 2022. [viewed date 2 May 2023]. Available from: `https://courses.fit.cvut.cz/NI-REV/media/lectures/rev01en.pdf`

[39] Odersky, M. Can We Wean Scala Off Implicit Conversions? In: *Scala Contributors* [online]. 2020. [viewed date 2 May 2023]. Available from: `https://contributors.scala-lang.org/t/can-we-wean-scala-off-implicit-conversions/4388`

[40] Scala Community. scala/scala-parser-combinators: simple combinator-based parsing for Scala. In: *GitHub* [online]. Jan 2023. [viewed date 2 May 2023]. Available from: `https://github.com/scala/scala-parser-combinators`

[41] Máj, P. TinyC AST Implementation (ast.h). In: *TinyC Project Repository – GitLab* [online]. 2023. [viewed date 2 May 2023]. Available from: `https://gitlab.fit.cvut.cz/NI-GEN/ni-gen-23/-/blob/main/tinycc/frontend/ast.h`

Appendix A

Acronyms

µC microC.

ALU arithmetic logic unit.

API application programming interface.

AST abstract syntax tree.

BIE-OOP Object-Oriented Programming.

BP base pointer.

CBC Common Bus Compiler.

CFG control-flow graph.

CISC complex instruction set computer.

CLI command-line interface.

CPU central processing unit.

DAG directed acyclic graph.

DFG data-flow graph.

DFS depth-first search.

DP dynamic programming.

DRY don't repeat yourself.

DSL domain-specific language.

EOF end of file.

GCC GNU Compiler Collection.

I/O input/output.

IDE integrated development environment.

IPT instructions per tick.

IR intermediate representation.

ISA instruction set architecture.

JIT just-in-time.

JNI Java Native Interface.

JVM Java Virtual Machine.

LLVM Low Level Virtual Machine.

NIE-APR Static Program Analysis.

NIE-GEN Compiler Construction.

OS operating system.

PCC Portable C Compiler.

PEG parser expression grammar.

RAM random access memory.

RISC reduced instruction set computer.

RTL register transfer list.

SP stack pointer.

SSA single static assignment.

SWIG Simplified Wrapper and Interface Generator.

TCC Tiny C Compiler.

tiny86 Tiny x86.

tinyC tinyC.

TIP Tiny Imperative Programming.

VM virtual machine.

Appendix B

User Guide

This user guide briefly describes how to compile and use the modular tinyC compiler. The directory tinycc of the electronic attachment contains a copy of the source code repository.

B.1 Building

A prerequisite for running tests and compiled tiny86 programs is a working tiny86 interpreter.

B.1.1 Building Tiny86

The source code repository includes a modified version of the tiny86 VM with some bug fixes and support for a text-based assembly format in the t86 subdirectory. The parser and assembly syntax was designed by Filip Gregor as part of his work on an interactive tiny86 debugger.

Tiny86 is a standard CMake[11] project and can be compiled by running the t86/scripts/build.sh script. The main CLI binary is saved to t86/build/t86-cli/t86-cli.

B.1.2 Building the Compiler

This project uses sbt[12]. We provide scripts/setup_env.sh, which installs asdf[13] to the home directory and uses it to grab all the required Java, Scala and sbt versions[14]. If you already have asdf installed, you can do the same using asdf install.

[11] https://cmake.org/
[12] https://www.scala-sbt.org/
[13] A CLI tool to manage runtime versions – https://asdf-vm.com/
[14] Tested versions configured in .tool-versions: Java 11.0.18+10, Scala 2.13.10, sbt 1.8.2

To build the project, run scripts/build.sh or sbt assembly in the root directory of the repository. Unit and integration tests can be executed with sbt test.

We have tested the instructions above on Ubuntu 22.04 with build-essential, cmake, git and curl packages.

B.2 Usage

The compiler is controlled through the CLI exposed as the tinycc wrapper script, which contains a built-in help available through `tinycc help`.

B.2.1 Compiling and Running TinyC Source

```
# create hello.c with an example program
cat >hello.c <<EOF
int main() {
  char *str = "Hello,␣world!\n";
  while(*str) {
    print(*(str++));
  }
  return 0;
}
EOF

# compile hello.c w/o optimizations
./tinycc compile -o hello.t86 hello.c

# compile hello.c w/ optimizations
./tinycc compile -O -o hello.t86 hello.c

# run hello.t86 on tiny86
./t86/build/t86-cli/t86-cli run hello.t86
```

Instead of directly compiling to assembly, we can run the frontend, middleend and backend separately:

```
# compile to IR
./tinycc compile-to-ir -o hello.ir hello.c

# optimize the IR
./tinycc optimize -o hello.opt.ir hello.ir

# compile the optimized IR to assembly
./tinycc codegen -o hello.t86 hello.opt.ir
```

The IR grammar is described in appendix C.

B.2.2 Additional Options

Most subcommands accept --verbose and --profile options. The former enables additional logging to the standard error output, the latter displays statistics about time spent in individual components of the compiler.

The number of available integer and float registers is 4, respectively 5 and it is configurable with the --register-cnt=M and --float-register-cnt=M options passed to compile or codegen actions.

Do not forget to configure tiny86 with the corresponding register count via -registerCnt=N and -floatRegisterCnt=M passed to t86-cli (default is 10 and 5). The size of RAM can be configured with -ram=S (default is 1024 words). We have also added -stats option to t86-cli, which prints number of elapsed ticks and executed instructions by the VM.

```
# compile with 2 integer and 2 float registers
./tinycc compile --register-cnt=2 --float-register-cnt=2 -O -o
    ↪ hello.t86 hello.c

# run with 2 integer, 2 float registers and 128 words of RAM
./t86/build/t86-cli/t86-cli run -registerCnt=2 -floatRegisterCnt
    ↪ =2 -ram=128 -stats hello.t86
```

B.2.3 Transpiling TinyC to C

For debugging purposes, the compiler contains a builtin tinyC to C transpiler, which can be used to execute tinyC programs directly on the host machine. A small runtime defining the builtin functions is included in src/test/resources/gcc_runtime.{c,h}. This is used by the run_tests.sh script to verify correctness of the reference output.

```
# compile the runtime
gcc --std=c99 -fsigned-char -c src/test/resources/gcc_runtime.c
    ↪ -o gcc_runtime.o

# transpile the tinyC program to C
./tinycc transpile-to-c --prefix='#include␣"gcc_runtime.h"' -o
    ↪ hello.transpiled.c hello.c

# compile the generated C program and link it with the runtime
gcc --std=c99 -I src/test/resources -fno-builtin -fsigned-char
    ↪ hello.transpiled.c gcc_runtime.o -o hello

# run the binary on the host system
```

```
./hello
```

B.3 Supported TinyC Features

The compiler supports all tinyC features from the language reference [3].

- `void`, `char`, `int`, `double`, pointers (including pointers to functions), static 1D arrays and structs
- basic arithmetic operators
- function calls, including recursion
- reading and writing a single character from stdin/stdout via `scan()` and `print(c)` builtins (`scan()` returns -1 on EOF)

In addition, the compiler supports the `printnum(n)` builtin, which prints an integer terminated by newline.

The file `examples/stdlib.c` contains some useful functions that you can copy into your programs (there is no preprocessor).

Appendix C

IR Grammar

⟨*letter*⟩ ::= (‘a’ | . . . | ‘z’) | (‘A’ | . . . | ‘Z’)
⟨*digit*⟩ ::= ‘0’ | . . . | ‘9’
⟨*identifier-start*⟩ ::= ⟨*letter*⟩ | ‘$’ | ‘_’
⟨*identifier-mid*⟩ ::= ⟨*identifier-start*⟩ | ⟨*digit*⟩
⟨*identifier*⟩ ::= ⟨*identifier-start*⟩ { ⟨*identifier-mid*⟩ }
⟨*register*⟩ ::= ‘%’ ⟨*identifier-mid*⟩ { ⟨*identifier-mid*⟩ }
⟨*int-part*⟩ ::= ⟨*digit*⟩ { ⟨*digit*⟩ }
⟨*frac-part*⟩ ::= ‘.’ ⟨*digit*⟩ { ⟨*digit*⟩ }
⟨*exp-part*⟩ ::= (‘e’ | ‘E’) [‘+’ | ‘-’] ⟨*digit*⟩ { ⟨*digit*⟩ }
⟨*integer*⟩ ::= [‘-’] ⟨*int-part*⟩
⟨*double*⟩ ::= [‘-’] ⟨*int-part*⟩ [⟨*frac-part*⟩] [⟨*exp-part*⟩]

⟨*insn-ref*⟩ ::= ‘null’ | ⟨*register*⟩
⟨*basic-block-ref*⟩ ::= ‘null’ | (‘label’ ⟨*register*⟩)
⟨*fun-ref*⟩ ::= ‘null’ | ⟨*identifier*⟩

⟨*program*⟩ ::= { ⟨*fun-decl*⟩ }
⟨*fun-decl*⟩ ::= ‘fn’ ⟨*ret-type*⟩ ⟨*identifier*⟩ ‘(’ ⟨*arg-type*⟩ { ‘,’ ⟨*arg-type*⟩ } ‘)’ ‘{’ ⟨*fun-body*⟩ ‘}’
⟨*fun-body*⟩ ::= ⟨*basic-block*⟩ { ⟨*basic-block*⟩ }
⟨*basic-block*⟩ ::= ⟨*identifier*⟩ ‘:’ { ⟨*insn*⟩ }
⟨*insn*⟩ ::= ⟨*register*⟩ ‘=’ (⟨*iimm*⟩ | ⟨*fimm*⟩ | ⟨*binary-arith*⟩ | ⟨*cmp*⟩
| ⟨*allocg*⟩ | ⟨*allocl*⟩ | ⟨*load*⟩ | ⟨*store*⟩ | ⟨*getelementptr*⟩ | ⟨*sizeof*⟩
| ⟨*getfunptr*⟩ | ⟨*loadarg*⟩ | ⟨*call*⟩ | ⟨*callptr*⟩ | ⟨*putchar*⟩
| ⟨*putnum*⟩ | ⟨*getchar*⟩ | ⟨*phi*⟩ | ⟨*cast*⟩ | ⟨*ret*⟩ | ⟨*retvoid*⟩
| ⟨*halt*⟩ | ⟨*br*⟩ | ⟨*condbr*⟩)

⟨*iimm*⟩ ::= ‘`iimm`’ ⟨*integer*⟩
⟨*fimm*⟩ ::= ‘`fimm`’ (⟨*integer*⟩ | ⟨*double*⟩)
⟨*binary-arith*⟩ ::= (‘`iadd`’ | ‘`isub`’ | ‘`iand`’ | ‘`ior`’ | ‘`ixor`’ | ‘`ishl`’
| ‘`ishr`’ | ‘`umul`’ | ‘`smul`’ | ‘`udiv`’ | ‘`sdiv`’ | ‘`fadd`’
| ‘`fsub`’ | ‘`fmul`’ | ‘`fdiv`’) ⟨*insn-ref*⟩ ‘`,`’ ⟨*insn-ref*⟩
⟨*cmp*⟩ ::= (‘`cmpieq`’ | ‘`cmpine`’ | ‘`cmpult`’ | ‘`cmpule`’ | ‘`cmpugt`’
| ‘`cmpuge`’ | ‘`cmpslt`’ | ‘`cmpsle`’ | ‘`cmpsgt`’ | ‘`cmpsge`’
| ‘`cmpfeq`’ | ‘`cmpfne`’ | ‘`cmpflt`’ | ‘`cmpfle`’ | ‘`cmpfgt`’
| ‘`cmpfge`’) ⟨*insn-ref*⟩ ‘`,`’ ⟨*insn-ref*⟩

⟨*allocl*⟩ ::= ‘`allocl`’ ⟨*var-type*⟩
⟨*allocg*⟩ ::= ‘`allocg`’ ⟨*var-type*⟩ [‘`,`’ ⟨*integer*⟩ { ⟨*integer*⟩ }]
⟨*load*⟩ ::= ‘`load`’ ⟨*scalar-type*⟩ ⟨*insn-ref*⟩
⟨*loadarg*⟩ ::= ‘`loadarg`’ ⟨*integer*⟩
⟨*store*⟩ ::= ‘`store`’ ⟨*insn-ref*⟩ ‘`,`’ ⟨*insn-ref*⟩
⟨*getelementptr*⟩ ::= ‘`getelementptr`’ ⟨*var-type*⟩ ‘`,`’ ‘`ptr`’ ⟨*insn-ref*⟩ ‘`,`’
‘`[`’ ⟨*insn-ref*⟩ ‘`]`’ ‘`.`’ ⟨*integer*⟩
⟨*sizeof*⟩ ::= ‘`sizeof`’ ⟨*var-type*⟩

⟨*getfunptr*⟩ ::= ‘`getfunptr`’ ⟨*fun-ref*⟩
⟨*call*⟩ ::= ‘`call`’ ⟨*ret-type*⟩ ⟨*fun-ref*⟩ ‘`(`’ ⟨*call-args*⟩ ‘`)`’
⟨*callptr*⟩ ::= ‘`callptr`’ ⟨*ret-type*⟩ ⟨*insn-ref*⟩ ‘`(`’ ⟨*call-args*⟩ ‘`)`’
⟨*call-args*⟩ ::= [⟨*arg-type*⟩ ⟨*insn-ref*⟩ { ‘`,`’ ⟨*arg-type*⟩ ⟨*insn-ref*⟩ }]
⟨*putchar*⟩ ::= ‘`putchar`’ ⟨*insn-ref*⟩
⟨*putnum*⟩ ::= ‘`putnum`’ ⟨*insn-ref*⟩
⟨*getchar*⟩ ::= ‘`getchar`’
⟨*phi*⟩ ::= ‘`phi`’ ‘`[`’ ⟨*insn-ref*⟩ ‘`,`’ ⟨*basic-block-ref*⟩ ‘`]`’ { ‘`,`’ ‘`[`’
⟨*insn-ref*⟩ ‘`,`’ ⟨*basic-block-ref*⟩ ‘`]`’ }

⟨*ret*⟩ ::= ‘`ret`’ ⟨*insn-ref*⟩
⟨*retvoid*⟩ ::= ‘`retvoid`’
⟨*halt*⟩ ::= ‘`halt`’
⟨*br*⟩ ::= ‘`br`’ ⟨*basic-block-ref*⟩
⟨*condbr*⟩ ::= ‘`condbr`’ ⟨*insn-ref*⟩ ⟨*basic-block-ref*⟩ ⟨*basic-block-ref*⟩
⟨*cast*⟩ ::= (‘`bitcastint64todouble`’ | ‘`sint64todouble`’
| ‘`bitcastdoubletoint64`’ | ‘`doubletosint64`’) ⟨*insn-ref*⟩

⟨*scalar-type*⟩ ::= ‘`i64`’ | ‘`double`’ | ‘`ptr`’

⟨*arg-type*⟩ ::= ⟨*scalar-type*⟩

⟨*ret-type*⟩ ::= ⟨*scalar-type*⟩ | ‘`void`’

⟨*var-type*⟩ ::= (⟨*scalar-type*⟩ | ⟨*struct-type*⟩) { ‘`[`’ ⟨*integer*⟩ ‘`]`’ }

⟨*struct-type*⟩ ::= ‘`struct`’ ‘`{`’ ⟨*var-type*⟩ { ‘`,`’ ⟨*var-type*⟩ } ‘`}`’

Appendix D

Contents of the Electronic Attachment

- `benchmarks` ....source code and results used for evaluation
- `latex` ....the directory of LaTeX source code of the thesis
- `tinycc` ....the Git repository with source code of the compiler including prebuilt binary target/scala-2.13/tinycc.jar
  - `t86` .....the Git repository with source code of the modified tiny86 VM including prebuilt binary build/t86-cli/t86-cli
- `CONTENTS.md` .... description of the attachment contents
- `thesis.pdf` ....the thesis text in PDF format