IMPROVING COMPILER CONSTRUCTION EDUCATION BY RETARGETING AND EXTENDING A COMPILER FOR EMBEDDED XINU

by

Liam M. Murphy, B.S.

A Thesis Submitted to the Faculty
of the Graduate School, Marquette University,
in Partial Fulfillment of the Requirements for
the Degree of Master of Science

Milwaukee, Wisconsin

May 2022

ABSTRACT
IMPROVING COMPILER CONSTRUCTION EDUCATION BY RETARGETING AND EXTENDING A COMPILER FOR EMBEDDED XINU

Liam M. Murphy, B.S.

Marquette University

Compilers are challenging to write and compiler construction is even more challenging for students to learn. Compiler construction courses are offered at many universities worldwide, but a trend has emerged among students learning about compilers for the first time. Compiler phases like lexing and parsing tend to be well grasped by students, but later concepts like type checking, translation, and register allocation are weak points, which cause many students to perform poorly. This work presents a useful tool for teaching compiler construction to students who are already familiar with operating systems concepts.

The Concurrent MiniJava compiler is now compatible with the latest version of the Embedded Xinu kernel. These two elements combined provide Marquette University computer science students the opportunity to directly engage with concepts learned in previous coursework to help improve and increase their understanding of compiler construction. This work is also applicable to students in other courses, such as operating systems and computer security, where assembly language is a major topic of discussion.

ACKNOWLEDGMENTS

Liam M. Murphy, B.S.

I would like to thank the following people, without whom this work would not have been possible:

- My parents, Timothy and Barbara, for instilling in me the importance of an education and for their unconditional support
- My advisor and friend, Dr. Dennis Brylow, for his guidance and advice during this project and throughout my time at Marquette
- My committee members, Dr. Debbie Perouli, Dr. Praveen Madiraju, and Dr. Sabirat Rubya, for their willingness to review and scrutinize my work
- My Evans Scholars friends, who have been very supportive and encouraging during my graduate studies

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
Introduction

1.1 Thesis Statement

A suitable educational compiler, supporting a new architecture, access modifiers, and multi-core concurrency, can be developed from an existing compiler infrastructure for use with the Embedded Xinu kernel within a multi-core Raspberry Pi environment, while retaining the pedagogical simplicity of the original design.

1.2 Overview

There exists a gap in today's compiler construction education. At Marquette University, the compiler construction course is taught using a version of an educational compiler targeting the MIPS architecture. For the benefit of undergraduate and graduate students, it is important that the compiler construction course be updated to reflect the latest technologies taught in the rest of the computer science curriculum. A useful tool for this task is the Embedded Xinu operating system, used in several courses within the computer science course catalog. Marquette students studying computer science have a unique educational experience in that they have a continuous thematic curriculum. Students progress from zero programming experience to learning basic data structures, to implementing their own embedded operating system running on real multi-core hardware, to finally implementing their own object-oriented language compiler that runs under the operating system built in a previous course. This research presents a tool to create vast learning opportunities for compiler construction students as well as students studying operating systems and computer security.

The current discourse within the compiler construction education community includes an identification of one main problem and several potential solutions. The main problem is that concepts in compiler construction are difficult for many students to learn. It is easy for a student to understand the concept of a compiler, in that it translates source code into machine code, but many of the more complicated ideas

necessary for a compiler to function do not get understood as easily. The existing methods that instructors use to teach compiler construction have varying degrees of success. Because of this, instructors have begun testing various techniques and tools of instruction to improve their students' understanding and increase their assessment scores.

1.3 Contributions

Building upon previous work on the Concurrent MiniJava compiler and existing computer science curriculum, we present an updated and extended compiler to improve student success in learning concepts in an upper-division compiler construction course. We have retargeted the Concurrent MiniJava compiler from the MIPS32 instruction set to ARMv7. This allows the compiler to become compatible with the latest version of the Embedded Xinu kernel for the Raspberry Pi 3B+ platform. We have also extended the MiniJava grammar to support the access modifiers, `public` and `private`, for variables and methods.

1.4 Organization

This thesis is organized as follows:

- Chapter 1 introduces the overall problem and our contributions to its solution.
- Chapter 2 introduces background on MiniJava and gives an overview on the phases of a compiler.
- Chapter 3 explores related work on MiniJava and other educational projects for compilers.
- Chapter 4 explains the effort to retarget the MiniJava compiler to support the ARMv7 architecture.
- Chapter 5 details how access modifiers have been added to the MiniJava grammar and the compiler error messages associated with them.
- Chapter 6 explains how MiniJava built-in functions have been implemented

within the Embedded Xinu kernel and the educational opportunities for undergraduate and graduate students.

- Chapter 7 briefly explains the challenges faced with this project and how we overcame them.
- Chapter 8 gives a summary of the contributions of this thesis and offers possibilities for future work.

CHAPTER 2
Background

2.1 MiniJava

MiniJava is a an educational programming language that is a subset of the Java programming language. Being a subset of Java, any MiniJava program can be compiled by an off-the-shelf Java compiler and executed on the Java Virtual Machine. The MiniJava compiler itself is written in Java and compiles input source files into the target assembly language, not Java bytecode. Appel and Palsberg's textbook [1] uses MiniJava to describe the underlying theory of each phase of the compiler and how to implement them. Thus, the MiniJava compiler is a suitable semester-long project for both undergraduate and graduate level courses. The original MiniJava implementation as described in [1] targets the 32-bit MIPS SPIM simulator, but is modular enough to allow for reimplementation to a variety of different platforms.

2.2 Concurrent MiniJava

Concurrent MiniJava is the most recent version of the MiniJava language used at Marquette University to teach undergraduate and graduate level Compiler Construction courses. The MiniJava compiler as described in [1] was extended to support concurrency features like threads and synchronized methods [2].

2.3 The Concurrent MiniJava Grammar

The Concurrent MiniJava language grammar is a strict subset of the standard Java grammar. Concurrent MiniJava supports only two primitive types: `int` and `boolean`; a `void` return type; and a string literal that can only be used as an argument to a print statement. There is no string primitive or `String` object. Concurrent MiniJava supports many of the expected features of Java, however many other features, such as garbage collection and floating point data types, are not supported in this implementation due to the amount of time needed for students to build out such features. Garbage collection and floating point data types may suitable topics for a second semester graduate-level compilers course.

```
<registration>  ::=  <prefix> <number> <suffix>
      <prefix>  ::=  "NCC" "-"
      <number>  ::=  <first_digit> <last_digits>
      <suffix>  ::=  "-" <letter>
                 |   ε
 <first_digit>  ::=  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  <last_digit>  ::=  <digit> <last_digits>
                 |   ε
       <digit>  ::=  <first_digit>
                 |   0
      <letter>  ::=  A | B | C | ...  | Z
```

Figure 2.1: Language of Starfleet registration numbers in pseudo Backus-Naur Form

2.3.1 Context-Free Grammars

Aho et al. [3] show that context-free grammars are defined by the following four properties:

- A set of terminal symbols, sometimes referred to as *tokens*
- A set of nonterminal symbols, sometimes referred to as *syntactic variables*.
- A set of productions, each production consisting of a nonterminal and a sequence of terminals and/or nonterminals.
- A nonterminal identified as the "start" symbol, sometimes referred to as a *goal*.

Let us consider an example of a context-free grammar: the language Starfleet starship registration numbers from *Star Trek*. It is common knowledge that the registration number of Captain James T. Kirk's U.S.S. Enterprise is *NCC-1701*. The observant fan will recognize that subsequent iterations of the famed starship have similar registrations, each a derivative of the original but with the addition of a suffix letter: *NCC-1701-A*, *NCC-1701-B*, *NCC-1701-C*, *NCC-1701-D*, etc.

In Figure 2.1, we show the language of Starfleet registration numbers. The "goal" of this grammar is a `registration`. Each `registration` is derived by `prefix`,

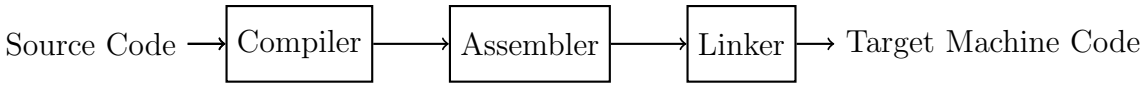

Figure 2.2: The Process of Language Translation

`number`, and `suffix` nonterminals. Both `suffix` and `last_digit` have epsilon derivations. In the case of context-free grammars, epsilon, ε signifies an empty string. One can think of it as the "absence" of that particular production. As an example, the language of Starfleet registration numbers would accept this string: "NCC-1701". In this example, there appears to be no suffix. Rather, we can understand this absense of a suffix to be the empty string, which is a valid derivation.

Another case to note is the `digit` nonterminal. A `digit` can be produced by a `first_digit` or a "0". Notice how we can re-use previous nonterminals in subsequent productions to simplify our grammar. We could easily define `digit` to contain the derivations "0", "1", ..., "9". This would be correct, however it would appear redundant.

2.4 The Phases of a Compiler

This discussion of context-free grammars is important to lay the foundation for how compilers work. The basic function of a compiler is to be a translator. Compilers translate from source code written in a high-level programming language to source code written in a lower-level language like C or assembly. The key to a successful compiler is the ability to preserve the *semantic meaning* of the original program after it has been translated into the target language.

Figure 2.2 shows the process of compilation at a high level. First, the source program is passed into the compiler. The compiler does some black-box magic and outputs target assembly code. Then, that assembly code gets passed to the assembler. The assembler does more black-box magic and creates target machine code. That machine code gets passed to the linker, where external libraries and object files are linked together into one executable target machine code file.

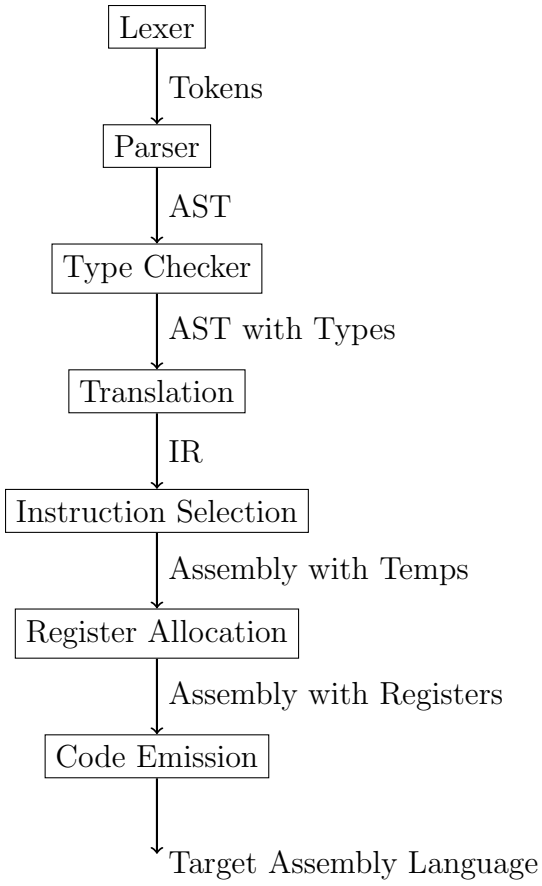

Figure 2.3: Compiler Stages

Modern compilers are not monolithic programs. Generally, they are split into several modules, or stages. Each stage is responsible for processing the program in some way until the same program in the target language emerges on the other end. Depending on the type of compiler, there are usually at least six stages of compilation: lexing, parsing, type checking, translation, instruction selection, register allocation, and code emission. Each stage may be broken down into more than one action. Some compilers may not implement some stages entirely. Figure 2.3 shows a flow diagram of the main stages of the compiler. We will discuss each of them in more detail.

2.4.1 The Lexer

The *lexer*, also called the *scanner* or *tokenizer*, is responsible for recognizing the tokens of a language. The lexer gets its name from the action it performs: *lexical analysis*. The easiest way to manually implement a lexer is through the use of regular expressions and finite state machines (really, you could implement regular expressions *as* finite state machines, or use a combination of the two).

The lexer addresses its input file one character at a time. This character stream gets passed through a set of regular expressions or state machines for each of the reserved tokens of the language. The set of tokens for a language encompasses both reserved words and symbols. Examples of MiniJava reserved words include `class`, `int`, `void`, `public`, and `static`. Examples of MiniJava symbols include mathematical symbols like `+`, `-`, and `*`. Other symbols include {, }, `(`, `)`, `&`, `&&`, and so on. In most programming languages, whitespace, areas in the source code that do not contain text, do not have any syntactic meaning. Whenever the lexer encounters whitespace symbols characters of strings, it ignores them and accepts the next token. The same must be said regarding comments. Many, if not all, modern programming languages have some syntax for handling comments. Some languages, like C, C++, and Java, allow for both single and multi-line comments. It is the lexer's job to recognize the beginning and ending markers for comments and to ignore all other characters between those markers. Other languages, like Python, only support single line comments. These can be easier to implement, as the lexer must only ignore characters until the end of a line.

Lexical analysis also involves deciding the initial types of variables. Lexers must be able to discern decimal integers from hexadecimal or octal ones. They can also be used to determine floating point values, strings, and booleans. Whether or not these types are *actually* correct given the variable they describe to cannot be determined by through lexical analysis. This will be done at a later phase of compilation.

The output of a lexer is the *token stream*. As soon as a lexer is able to assign a stream of characters a token name, it outputs that token to be accepted by the next phase: the parser. An example of the MiniJava token stream may look like this:

```
CLASS
ID(HelloWorld)
LBRACE
PUBLIC
STATIC
VOID
MAIN
LPAREN
...
```

Some design decisions can be made when writing a lexer. One could decide to treat `main` as either an identifier or a reserved word. One could decide that all identifiers are strings until determined otherwise by the parser. The Concurrent MiniJava compiler treats `main` as a reserved word and not an identifier, but does treat all other method names as the latter.

2.4.2 The Parser

The next phase of the compiler is the parser. Parsing is a complicated action that can be performed with a myriad of techniques. The Concurrrent MiniJava parser is a LL(1)[1] top-down recursive-descent parser generated by the JavaCC parser-generator tool [4]. We will discuss parser-generators in the last section of this chapter.

From the lexer, the token stream is fed into the parser where it is processed via *syntactic analysis*. Syntactic analysis is the operation that processes tokens according to the rules of a language's grammar and builds a structure called an *abstract syntax tree*. An abstract syntax tree represents the structure of a program's statements,

[1] In the best case, our parser parses expressions with a left-lookahead of 1. In some instances, the parser must look ahead more than one token, but the average case is still one.

but does not necessarily enforce the correct grammar rules of the source language. Abstract syntax trees must be structured in order to enforce operator precedence rules for how language statements get evaluated. They must also be able to detect erroneous statements and report those errors to the human programmer. The abstract syntax tree generated by a parser does not necessarily remain immutable. Later stages of the compiler will operate on the initial AST and store additional information regarding the contents of the program.

2.4.3 Type Checking

Type checking is the process of *semantic analysis*, which assigns meaning to the program represented by an abstract syntax tree. An abstract syntax tree can represent a syntactically correct program, however that program can be totally incorrect in its meaning. A famous example of a syntactically correct, yet semantically ridiculous structure is, "colorless green ideas sleep furiously." This was written by Noam Chomsky in his book *Semantic Structures* [5]. Chomsky's example is a grammatically correct English sentence, but it has no meaning. The same idea can be seen in many common programming errors that a human may encounter.

Consider a strongly typed language like Java. Suppose we try to write a Java program that assigns the string "pizza" to an `int`. While a variable assignment statement would be a legal construct according the grammar, the data types on the left and right hand sides of the statement would not agree with each other. Consider another example where a variable is being assigned before it is declared. This would also raise an error. Using a variable in a function call, for example, is permissible, however a variable cannot be referenced without having first been declared. During the type checking phase, the compiler builds a symbol table by scanning through the abstract syntax tree and creating entries for identifiers: class, function, and variable names. These identifiers become associated with data types which helps to resolve the types of expressions within statements. By the end of type checking, the abstract syntax tree will have been decorated with identifier types and the program will have been checked for any semantic errors. The product is a

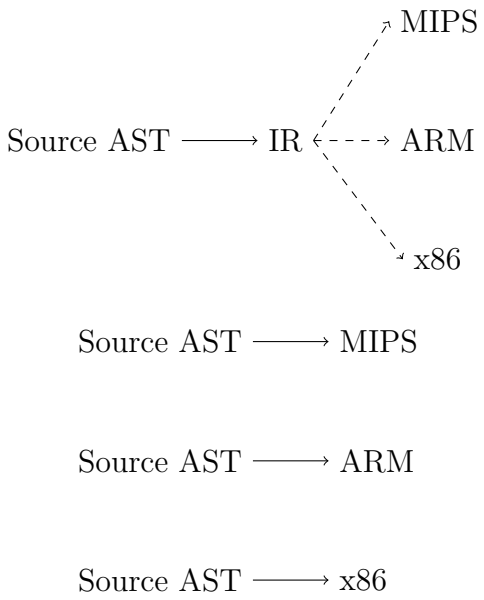

Figure 2.4: Intermediate Representation Simplifies Mapping to Target Architectures

grammatically correct program.

2.4.4 Translation

The translation phase consists of *mapping* abstract syntax tree nodes to nodes within the intermediate representation tree. Intermediate Representation can be thought of as an *abstraction* of a target assembly language, allowing compilers to retain a desirable quality of modularity, wherein a single compiler frontend can be used with any number of compiler backends. In other words, a compiler can process a **single** programming language into the machine language of **several** target architectures, as illustrated in Figure 2.4. Compilers do not need to implement this feature, however. In situations where a language's implementation must be specific to a particular architecture, it does not seem practical to design and implement the notion of an intermediate representation to which the abstract syntax tree is translated; the abstract syntax tree can be directly processed into assembly instructions.

2.4.5 Instruction Selection

Instruction selection involves deciding which assembly instructions are needed to complete the operation represented by a node within the intermediate representation tree. Typically, the first step is to organize the intermediate representation tree

into a linear list of chunks. Each chunk, called a basic block, encapsulates an independent block of instructions that do an action and branch to another block at the end of that action. Basic blocks must not contain any intermediate branch actions. Sometimes it is best to organize these blocks out of order, which can be a technique for increasing the speed of a program. As long as the blocks are properly constructed, each block will branch to the correct subsequent block, and so on until the program terminates.

Once the program has been organized into a series of basic blocks of intermediate representation nodes, we must overlay patterns of these nodes in order to work out which assembly code instructions get written. Each instruction in the instruction set corresponds to a pattern of intermediate representation nodes. For example, the ARM `add` instruction can either add the contents of two registers or it can add the contents of a register and some constant value. These two operations are represented by different patterns of IR tree nodes and use different assembly code syntax. This is the ultimate goal of instruction selection: determine the patterns and choose the appropriate instructions for each pattern.

An important abstraction of instruction selection is not to immediately determine which registers are used for any particular pattern. Each assembly instruction is written using a temporary value representing a register. Assembly code programs at this stage are generated with an unbounded number of temporaries that will get resolved into actual register names during the register allocation phase.

2.4.6 Register Allocation

Register allocation is the task of assigning the physical registers to each temporary. This task seems challenging because there are a potentially infinite number of temporaries and only a finite number of registers. We use techniques such as liveness analysis and graph coloring to determine the registers that need to be used in a specific order.

Liveness analysis is the process that decides which registers are in use through-

out portions of the program. The register allocator must know an architecture's calling convention for caller-save and callee-save registers so that the values within registers do not get overwritten at inappropriate times. Graph coloring is a technique that helps to determine the order in which registers can be used within a program. The pool of temporaries gets processed by the register allocator to produce assembly instructions that operate on the correct registers at the correct point in the lifetime of the program.

2.4.7 Code Emission

Code emission is the mechanism that actually writes out the lines of assembly language into a file. This occurs after the instructions are chosen and the correct ordering of physical registers is determined. After this final stage of the compiler, the finished product is the semantically correct assembly code representation of the compiler's source language.

2.5 Parser Generators

Parser generators, also called compiler compilers, are useful tools for processing input source code. These programs can execute the very mechanical task of lexical analysis and syntactic analysis. Writing lexers and parsers by hand can be very tedious and time consuming, so tools like JavaCC and Yacc can help automate the process and eliminate the chance for errors at early stages in the compiler.

Yacc (Yet Another Compiler Compiler) [6] is an example of a parser generator. Yacc takes a description of the source language grammar and converts it into a C program that generates an abstract syntax tree. JavaCC is a similar program that generates a recursive-descent parser written in Java. This program works well within the Concurrent MiniJava compiler because it helps students to focus more energy on the later, more challenging phases of the compiler project.

2.6 Summary of Background

In this chapter, we have examined an overview of fundamental ideas necessary in building a compiler. First, we introduced our source language, MiniJava, and its

concurrency support. Next, explained context-free grammars and how they can be used to describe programming languages. Finally, we gave summaries of each of the phases of the compiler and how they go about transforming human-facing source code into machine-facing assembly code that runs on a target platform.

CHAPTER 3
Related Work

3.1 Further Development

The Compiler Construction course at Marquette University is the second half of a two course sequence in Programming Languages. This course uses the MiniJava compiler to teach the fundamentals of compiler construction including topics such as classes of grammars, automata theory, and graph theory. By the end of the semester, students will have built a full end-to-end compiler that can output assembly code that executes on real multicore hardware instead of simulators.

3.2 Concurrent MiniJava

The previous iteration of the MiniJava compiler was completed in 2010 by Adam Mallen at Marquette University. Mallen's work brought Java-style threads into the MiniJava language and the Embedded Xinu kernel [2]. This two-part project had the goal of refactoring Marquette's Compiler Construction course in order to "increase student interest and motivation" for undertaking coursework where one builds a working compiler [2]. Mallen's efforts resulted in a MiniJava compiler that targets the MIPS implementation of the Embedded Xinu operating system, supported a wider range of I/O functionality (something that the original MiniJava language from Appel and Palsberg's textbook did not), and improvements to Embedded Xinu that supported multi-threaded object-oriented programming. For eleven years, Mallen's *Concurrent* MiniJava compiler was the tool used in teaching the Compiler Construction course at Marquette University.

The authors of [2] conducted both qualitative and quantitative assessments at the conclusion of a semester where Concurrent MiniJava was used during the the Fall 2009 Compiler Construction course. With the results of those assessments, the authors saw that student satisfaction increased compared to anecdotal testimonies collected from students in a section of the same course taught in 2007. Additionally, average scores for many weekly assignments increased between 2007 and 2009. This

supports the authors' hypothesis that using Embedded Xinu and concurrency features increases student success.

3.3 Bantam Java

Bantam Java is another compiler course project designed and implemented by Marc Corliss and E. Christopher Lewis [7]. Similar to MiniJava, Bantam Java is a subset of the Java language however, Bantam Java has several more features that MiniJava lacks. Bantam Java supports `int` and `boolean` types as well as native and non-native Java object types. Bantam Java also contains dynamic memory management via object instantiation with the `new` keyword and object deallocation using garbage collection. As with the MiniJava project, students become introduced to the Visitor pattern while implementing the components of the Bantam Java compiler.

The authors aimed to accomplish four goals: "carefully-selected features", "well-engineered infrastructure", "comprehensive documentation", and a "customizable project" [7]. Through these goals, the authors of Bantam Java offer flexibility to instructors teaching compiler construction courses. The Bantam Java project does not rely on any particular compiler construction textbook, nor must it remain immutable. The authors provide a full language manual along with a language runtime environment for x86 Linux machines. Through the compiler's modular and extensible design, instructors are given freedom to extend the language's features or omit certain compiler modules from their curriculum.

The authors write a second paper about their experiences teaching compiler construction [8]. They also share feedback from other instructors who have used Bantam Java at their institutions. Their feedback comes from six instructors at universities around the world that implemented the Bantam Java compiler in compiler courses. The instructors and students alike shared that, overall, using the Bantam Java compiler project helped students to learn critical compiler construction concepts. Some improvements were made to the compiler as a result of the first paper and of instructor feedback. The compiler was retargeted to the JVM platform. The grammar

of the language was extended to support arrays, for loops, increment and decrement operators, and others, in order to be suitable for graduate-level coursework.

3.4 Teaching Compilers

It is no secret that writing a compiler is a challenging enterprise. A compiler combines many aspects of computer science into one program, therefore, it can be difficult for undergraduate students to fully grasp each concept within a single semester. The goal of improving compiler construction education should be to make learning compilers as approachable as possible for both undergraduate and graduate students. There has been some work done in this area, however. Others at universities around the world have been putting new ideas into practice for helping students come to grips with the difficult concepts that make compilers work.

Some researchers have been studying different techniques for presenting the course material, other than the traditional lecture-based approach. Na Wang and Liping Li from the Shanghai Polytechnic University have designed a compiler construction course with a flipped-classroom approach [9]. The authors have argued that, through teaching reform, the students would be able to absorb more information in an environment where students have both online and in-person instruction [9]. The proposed hybrid teaching model includes situations where instructors provide in-person instruction along with small group discussions [9]. After the class meeting, students would take online assessments and participate in Q A and discussion activities [9]. The authors argue that such a teaching model "makes learning more flexible. If the learners want to acquire knowledge, they don't have to learn it at a designated place and time. Online resources provide learners more chances to choose what to learn and how to learn" [9].

Another approach involves group-based work. Joe Gibbs Politz and Yousef Alhessi at the University of California San Diego designed a course structure where the last four weeks of the semester were devoted to groups of graduate students each implementing separate components of a single compiler [10]. Each team consisted

of 2-3 students who were given four weeks to implement their chosen compiler component. For each component, its respective group had to collaborate with every other group to make sure that each component interfaced appropriately [10]. The instructors employed a technique they call mob programming. During mob programming sessions, the instructors led class discussion where designs were compared and groups were given the opportunity to provide suggestions [10]. After the four week period, each student completed an assignment that asked students to describe how they would implement an extension to their group's component [10]. The outcomes of these assignments were evaluated in one-on-one oral exams. The feedback collected by the instructors at the end of the course indicated that the students enjoyed the group-based design of the coursework, however the students felt that the amount of coursework was challenging. The authors remarked that future iterations of the course would involve a project six weeks in duration, rather than only four [10]. The authors stated, "based on our experience with mob programming and identifying key concepts through merging student work, we believe that many learning outcomes can be served through the project model without relying on individual assignments" [10].

John Lasseter from Hobart & William Smith Colleges, argues that using an interpreter to teach an undergraduate compiler construction course can help solidify certain compiler concepts [11]. Over the course of several years, Lasseter noticed a trend in how well students learn and understand compiler construction concepts. He explains, "students do well with the material on lexing and parsing, but too many of them begin to struggle during implementation of the semantic analysis phase. They find overwhelming both the conceptual material underlying type checking, the details of the visitor pattern in traversing an AST, and the interplay of this traversal technique with the complexities of a realistic semantic analysis implementation" [11]. His solution is to introduce an interpreter to explain the semantic analysis and code generation phases of the compiler.

Similar to the Compiler Construction course at Marquette University, Lasseter's course uses Andrew Appel's "Modern Compiler Implementation" textbook

[11]. Lasseter, however, uses the Tiger language as the source for his semester-long compiler project. He introduces the interpreter into the course schedule at the point where students have already implemented their own parsers. The interpreter is used to enforce ideas regarding traversing an abstract syntax tree, building a symbol table, and generating assembly code [11]. Rather than writing out JVM bytecode, Lasseter's interpreter sets the stage for code generation by showing students how to evaluate the left-hand side and right-hand side components of expressions. The author does acknowledge, however, that using interpreters to teach students how to write compilers does have its limitations. Nevertheless, Lasseter believes that this technique can be applied to other phases of the compiler pipeline, like data flow analysis, and machine code versus intermediate representation [11].

Doug Baldwin from SUNY Geneso designed a simple, modular compiler for use in teaching students within a single 14 week semester [12]. Baldwin notices a similar trend to that of Lasseter [12]. Baldwin notes that compiler projects are cumulative. Each phase builds upon the previous ones, and students who fail to completely understand the earlier concepts will inevitably fail at learning any subsequent concepts. His approach is to design a compiler for a language that is robust enough to provide opportunities for teaching classic compiler concepts. Baldin's language, called MinimL, supports operator precedence, subroutines, variables, and more [12]. These features are complex enough to generate interesting and challenging assignments that will help students understand how compilers work.

3.5 Summary of Related Work

In this chapter, we have discussed the previous work done to bring concurrency to the MiniJava language and Embedded Xinu kerenel. Next, we discussed the Bantam Java language, similar to MiniJava, and how it is being used to teach compiler construction. Lastly, have described some of the work being done by other computer science educators throughout the world to improve compiler construction education.

Na Wang and Liping Li from the Shanghai Polytechnic University designed

a flipped-classroom compilers course and showed that asynchronous learning can be beneficial for students learning the challenging topics involved with building compilers. Joe Gibbs Politz and Yousef Alhessi at the University of California San Diego explained their group-based approach to building a classroom compiler, where students in small groups collaborated to each build separate compiler modules. John Lasseter from Hobart & William Smith Colleges wrote an interpreter to help teach phases of the compiler that his students found most challenging to undeerstand. Doug Baldwin from SUNY Geneso wrote a compiler for his language called MinimL. Baldin showed that his compiler was robust, yet simple, allowing his students to learn compiler construction in a single semester.

It cannot be overstated that the ideas within compiler construction are complex and provide significant challenges to students today. Compiler projects are some of the most involved and complicated pieces of software that a student could write. Therefore, students can benefit from various techniques that can increase the odds for academic success.

CHAPTER 4
Retargeting from MIPS to ARM

4.1 RISC vs. CISC

In the world of processor design, there are two main categories of architecture: RISC and CISC. RISC processors (Reduced Instruction Set Computer) and their counterparts, CISC processors(Complex Instruction Set Computer), have different benefits that make them appropriate for one situation or another. RISC architectures have the benefit of implementing the minimum set of instructions that allow for arbitrary computation. Most of these operations need only use one or a small few clock cycles to complete [13]. This allows for quick, low powered computation, and a human programming experience that is easy to learn in undergraduate and graduate courses [14]. RISC processors are a popular choice for embedded systems like routers [15] and low-power, low-profile devices, like smartphones and single board computers like the Raspberry Pi [16].

CISC architectures implement a set of instructions that perform more advanced tasks and may require more than one clock cycle to complete. In the case of the x86 architecture, instructions can be of variable width. This is in contrast to RISC instructions, which are of uniform width. CISC architectures, like Intel x86, for example, pass arguments via the stack with a series of `push` and `pop` operations. RISC architectures, like ARM, pass variables via callee and caller save registers.

4.1.1 Comparing x86 and ARMv7 Assembly Languages

The classic first program for a beginner to write is "Hello, World!" While being a good test case to ensure a language's development and runtime environments are working correctly, it is also a nice example to compare the languages that CISC and RISC machines speak.

We have written an implementation of "Hello, World!" in the C programming language and compiled it on an x86-based Linux machine with both the x86 GCC C compiler and the ARMv7 GCC C cross-compiler. In Figures 4.1 and 4.2, we see the

```
1       .file   "helloworld.c"
2       .section    .rodata
3   .LC0:
4       .string "Hello, world!"
5       .text
6       .globl  main
7       .type   main, @function
8   main:
9   .LFB2:
10      .cfi_startproc
11      pushq   %rbp
12      .cfi_def_cfa_offset 16
13      .cfi_offset 6, -16
14      movq    %rsp, %rbp
15      .cfi_def_cfa_register 6
16      subq    $16, %rsp
17      movl    %edi, -4(%rbp)
18      movq    %rsi, -16(%rbp)
19      movl    $.LC0, %edi
20      call    puts
21      movl    $0, %eax
22      leave
23      .cfi_def_cfa 7, 8
24      ret
25      .cfi_endproc
26  .LFE2:
27      .size   main, .-main
28      .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-44)"
29      .section    .note.GNU-stack,"",@progbits
```

Figure 4.1: "Hello, World!" in x86 Assembly

output of running the GCC compiler with the `-S` flag, which outputs the resulting assembly language into a `helloworld.s` file. The source program listing can be found in Appendix A.

The x86 compilation output was 29 lines in length. On the eleventh line of code, we see the stack frame being setup by pushing the previous base pointer to the stack. Then, we see the address stored in the stack pointer being moved into the base pointer. Finally, we see the stack pointer being subtracted by 16, the calculated size of the stack frame. Lines 11-14 of this program show the stack frame being allocated in memory. The next two lines show space being allocated for the two formal parameters of the main function, `argv` and `argc`, and being stored in memory with offsets from the base pointer. The next line shows a `movl` instruction moving data at the `.LC0`

label, declared at the top of the file, into register `edi`. The string `"Hello, World!"` is stored at that memory location as a word. The next line calls the `puts()` function to print out the string. The final four lines show placing the return value `0` into a register to be returned, and returning from the function `main`.

In our source program, we decided to call `printf()` instead of `puts()`. Why did the compiler choose to call `puts()`? This was an optimization of the GCC compiler. It is likely that the C standard library implements `printf()` as an abstraction above the Linux system call `puts()`, so the compiler decided to call `puts()` directly.

The ARM program is similar to the x86 implementation, but differs in a number of important ways. Upon first glance, a programmer unfamiliar with the ARM instruction set would be able to glean the general flow of the program without too much study. The ARM opcodes follow a similar convention to their x86 counterparts. By contrast, the ARM register names are simpler than x86, omitting any special symbols in favor of alphanumeric characters only.

The ARM de-compilation output a total of 45 lines in length. The first nine lines contain build directives to assist the compiler in linking object files [17]. Lines 11-13 contain the file name, the section delimiter, and the word alignment. Lines 14-22 contain the label `.LC0` for the `"Hello world!"` string. The main function's implementation begins on line 23. To start, we see that the frame pointer and link register registers, `fp` and `lr`, are being pushed to the stack. Next, 4 is added to the address stored within the stack pointer and the result is stored within the frame pointer. In a later section of this chapter, we will discuss the significance of this line of the function prologue. The third line shows 8 being subtracted from the stack pointer. Like x86, this is the calculated size of the stack frame. Next, we see data being stored in registers `r0` and `r1` from memory locations offset by 8 and 12 from the frame pointer. Like in x86, `r0` and `r1` now contain the formal parameters `argv` and `argc`. Next, a word from memory is loaded into `r0`. The compiler has decided to store a reference to the label `.LC0` within the label `.L3`. Again, like the x86 program,

```
1       .cpu arm7tdmi
2       .eabi_attribute 20, 1
3       .eabi_attribute 21, 1
4       .eabi_attribute 23, 3
5       .eabi_attribute 24, 1
6       .eabi_attribute 25, 1
7       .eabi_attribute 26, 1
8       .eabi_attribute 30, 6
9       .eabi_attribute 34, 0
10      .eabi_attribute 18, 4
11      .file   "helloworld.c"
12      .section    .rodata
13      .align  2
14  .LC0:
15      .ascii  "Hello, world!\000"
16      .text
17      .align  2
18      .global main
19      .syntax unified
20      .arm
21      .fpu softvfp
22      .type   main, %function
23  main:
24      @ Function supports interworking.
25      @ args = 0, pretend = 0, frame = 8
26      @ frame_needed = 1, uses_anonymous_args = 0
27      push    {fp, lr}
28      add fp, sp, #4
29      sub sp, sp, #8
30      str r0, [fp, #-8]
31      str r1, [fp, #-12]
32      ldr r0, .L3
33      bl  puts
34      mov r3, #0
35      mov r0, r3
36      sub sp, fp, #4
37      @ sp needed
38      pop {fp, lr}
39      bx  lr
40  .L4:
41      .align  2
42  .L3:
43      .word   .LC0
44      .size   main, .-main
45      .ident  "GCC: (Fedora 7.1.0-5.el7) 7.1.0"
```

Figure 4.2: “Hello, World!” in ARMv7 Assembly

	MIPS	ARM
Total Registers	32	16
Caller Save (non-volatile)	12	4
Callee Save (volatile)	10	7
Word Size	4 bytes (32 bits)	4 bytes (32 bits)

Table 4.1: MIPS and ARM Architecture Specifications

the compiler has chosen to call the `puts()` system call in lieu of `printf()`. The following two `mov` instructions place 0 into register `r0` to be returned at the bottom of the function. The function epilogue contains instructions that destroy the stack frame by moving the stack pointer, popping the frame pointer and link register off of the stack, and branching to the address stored in the link register.

4.2 MIPS vs. ARM: An Overview

MIPS, specifically MIPS32, is a RISC (Reduced Instruction Set Computer) architecture that contains 32 general purpose registers and 32 floating point registers. MIPS processors are used in general-purpose computers like desktops, but are also used in embedded applications [18]. Table 4.1 compares architecture specifications between the MIPS and ARM 32-bit architectures. While the MIPS architecture contains more registers for the compiler to allocate for use in a program, the ARM architecture can implement the same programs using half of the number of registers. This is not to say that ARM-targeted compilers produce smaller binaries or that MIPS processors can execute programs faster. We are simply making the point that processors can be designed such that they do not need to contain an identical number of registers from architecture-to-architecture or brand-to-brand.

4.2.1 Comparing MIPS and ARM Stack Frames

The *stack frame*, also called an *activation record*, is an area in memory that contains the the local variables and instructions for a function call. In any program, there may be situations where a function could be called just once or perhaps several times. In either situation, each time a function is called, a new stack frame is pushed onto the call stack of the program. As a corollary, this is the reason that recursive functions, written in imperative, block-based languages, are expensive to

run. Each initial invocation of a recursive function, depending on its implementation, may create an exponential number of stack frames before returning the final result. Modern operating systems implement stack protection mechanisms to prevent a running program from exceeding a certain amount of stack space. If a running program exceeds its allotted block of stack space, a stack overflow error can be triggered and the program will crash.

Each architecture implements a different convention for stack frames. As we discussed earlier in this chapter CISC and RISC architectures are rather different. As such, they have different stack frame conventions. Since MIPS and ARM are both RISC-based architectures, their stack frames are slightly different, but are largely semantically similar.

Figure 4.3 shows the output from compiling a simple "Hello World" program with the Concurrent MiniJava compiler for the MIPS platform. Figure 4.4 shows the output from the same program compiled with the Concurrent MiniJava compiler for ARM. At first glance, we notice that the MIPS version is considerably longer than its ARM counterpart. However, the structure of each program is still the same. We have been able to keep the same labeling scheme between both architectures. The noticeable differences are in the opcode names and syntax, the function calling conventions, and lack of `push`/`pop` notation in the MIPS program. Another glaring difference is how many registers the MIPS compiler uses in the stack frame prologue and epilogue. The MIPS program makes three calls to values in memory to store and load at the beginning and end of the stack frame. This action gets replaced by the `push`/`pop` notation in the ARM implementation. This reduction in complexity can be helpful to students who are still uncomfortable with reading programs written in assembly language. It can be non-intuitive to dissect how words are being loaded and stored from offsets to other locations in memory. While this concept still exists in more complex programs compiled to ARM, every effort to redeuce student confusion will help them with concept comprehension in the long term.

```
1  #include <mips.h>
2          .data
3  L0:     .asciiz "Hello␣world!\n"
4          .text
5          .align  4
6          .globl  main
7  main:
8  main_framesize=16
9          addiu   sp,     sp,     -main_framesize
10         sw      ra,     -12+main_framesize(sp)
11         sw      s0,     -8+main_framesize(sp)
12         sw      s1,     -4+main_framesize(sp)
13         move    s1,     s2
14         move    s0,     s3
15         move    a3,     s4
16         addiu   v0,     sp,     main_framesize
17         move    a2,     v0
18 L2:
19         la      a0,     L0
20         jal     _print
21         // Call sink
22         move    v0,     zero
23 L1:
24         addiu   v1,     sp,     main_framesize
25         move    v1,     a2
26         move    s4,     a3
27         move    s3,     s0
28         move    s2,     s1
29         lw      s1,     -4+main_framesize(sp)
30         lw      s0,     -8+main_framesize(sp)
31         lw      ra,     -12+main_framesize(sp)
32         // return from main
33         addiu   sp,     sp,     main_framesize
34         jr      ra
```

Figure 4.3: “Hello World!” Output From the MIPS Concurrent MiniJava Compiler

```
1  #include <arm.h>
2          .section .data
3  L0:     .asciz  "Hello world!\n"
4          .section .text
5          .align  4
6          .globl  main
7  main:
8  #define main_framesize 8
9          push    { fp, lr }
10         add     fp,     sp,     #4
11         sub     sp,     sp,     #main_framesize
12 L2:
13         ldr     r0,     =L0
14         bl      _print
15         // Call sink
16         mov     r0,     #0
17 L1:
18         // return from main
19         sub     sp,     fp,     #4
20         pop     { fp, lr }
21         bx      lr
```

Figure 4.4: "Hello World!" Output From the ARM Concurrent MiniJava Compiler

4.3 Implementing an ARM Backend

The existing Concurrent MiniJava compiler architecture lends itself well to being retargeted. Since the compiler implements an intermediate representation, we can borrow the structure of our ARM implementation from the existing MIPS implementation.

4.3.1 Directory Structure

The back-end directory structure relies on two main modules: the Frame package and the Arm package. In Figure 4.5, we can see the similarities between the Mips and Arm packages. The main driver class of the compiler must explicitly choose which back-end to use when translating the type-checked abstract syntax tree into the intermediate representation. In the translation phase, the implementation of some abstract syntax tree nodes must call functions declared within the compiler's representation of a stack frame. As previously discussed, each architecture implements a different calling convention from one another and makes different decisions for how assembly code is written. At translation time, we must decide whether we want to

```
Frame/                  Mips/                   Arm/
     /Access.java            /InFrame.java           /InFrame.java
     /Frame.java             /InReg.java             /InReg.java
                             /MipsFrame.java         /ArmFrame.java
                             /Codegen.java           /Codegen.java
```

Figure 4.5: Back-end Directory Structure

compile our intermediate representation to MIPS or ARM assembly. Thanks to Java, our implementation language, we can use abstraction and inheritance to make our implementation easier.

4.3.2 Code Reuse

The Frame package contains a `Frame` abstract class that contains abstract method headers that an architecture-specific stack frame package must implement. The `Access` abstract class represents . This design allows for an extensible back-end suite and allows the compiler to support multiple architectures concurrently without needing to recompile.

In either the Mips or Arm packages, the most important files are `MipsFrame.java`, `Codegen.java`, `ArmFrame.java`, and `Codegen.java` respectively. The `MipsFrame` and `ArmFrame` classes contain the implementation of a stack frame for that particular architecture. The ARM stack frame implementation contains a constant for the word size of the architecture, arrays of `Temp`s (temporaries) for each register and whether they are special (frame pointer, stack pointer, link register, etc.), argument registers, caller/callee save, and colorable. Some of the important helper methods found within this class are `externalCall()`, `assignCallees()`, and `procEntryExit()`.

Figure 4.6 shows the implementation of the `externalCall()` method within the ArmFrame class. This method allows the code generator to call Xinu system calls within MiniJava programs. Xinu system calls are prefixed by an underscore. For example, the `externalCall()` is invoked when a programmer calls `Xinu.print()` or

```
1 public Tree.Exp externalCall(Symbol s, List<Tree.Exp> args) {
2     Label l = labels.get(s);
3     if (l == null) {
4         l = new Label("_" + s.toString());
5         labels.put(s, l);
6     }
7     return new Tree.CALL(new Tree.NAME(l), args);
8 }
```

Figure 4.6: `externalCall()` Definition in `ArmFrame`

when a programmer writes a `synchronized`. Both instances map to a Xinu system call defined in the `minijava.c` API.

Figure 4.7 shows the implementation of the assignFormals() and assignCallees() methods within the ArmFrame class. Since they are private helper functions, they cannot be called outside of the ArmFrame class. Both assignFormals() and assignCallees() recursively resolve either formal parameters or callee-save registers for an ARM stack frame. A particularly interesting case is the behavior of `assignCallees()`. When `assignCallees()` is called, it treats each callee-save register as a "live" register and moves them into temporaries within each stack frame. It is the job of the register allocator to determine which callee-save, if any, is redundant or unused and to remove it from the internal representation before emitting the final assembly code. A more sophisticated implementation of a stack frame might keep track of the callee-save registers for each frame and allocate the correct number before the register allocator is invoked.

Figure 4.8 lists the method that gets called in the first stage of setting up the prologue of a stack frame. The body of the method is passed in and statements that handle the formal parameters and callee-save registers are added to it. In the main driver file for the compiler, `procEntryExit1()` is called during the code generation phase. It is at this point when the stack frame begins to take shape. `procEntryExit2()` and `procEntryExit3()` are invoked during the register allocation phase. After code generation, each statement in the stack frame operates on temporary registers. The job of `procEntryExit2()` is to mark the return point of

```
1   private void assignFormals(Iterator<Access> formals, Iterator
        <Access> actuals, List<Tree.Stm> body) {
2       if (!formals.hasNext() || !actuals.hasNext()) return;
3
4       Access formal = formals.next();
5       Access actual = actuals.next();
6
7       assignFormals(formals, actuals, body);
8
9       if (formal != actual)
10          body.add(0, MOVE(formal.exp(TEMP(FP)), actual.exp(
                TEMP(FP))));
11  }
12
13  private void assignCallees(int i, List<Tree.Stm> body) {
14      if (i >= calleeSaves.length)
15          return;
16
17      Access a = allocLocal();
18
19      assignCallees(i + 1, body);
20
21      body.add(0, MOVE(a.exp(TEMP(FP)), TEMP(calleeSaves[i])));
22      body.add(MOVE(TEMP(calleeSaves[i]), a.exp(TEMP(FP))));
23  }
```

Figure 4.7: `assignFormals()` and `assignCallees()` in `ArmFrame`

```
1   public void procEntryExit1(List<Tree.Stm> body) {
2       assignFormals(formals.iterator(), actuals.iterator(),
            body);
3       assignCallees(0, body);
4   }
```

Figure 4.8: `procEntryExit1()` Definition in `ArmFrame`

a function for the human programmer with a comment. This helps the programmer to easily identify each point where a function may return control to the rest of the program. `procEntryExit3()` has the bigger role of creating the function prologue and epilogue. The function prologue contains the code text section header information, the stack frame size, and the function label. The most operative sections of the prologue are the lines that allocate space on the stack for the frame and stack pointers. The function epilogue contains the instructions for deallocating the space for the stack frame and branching to the instruction immediately after the call to the function. The implementations for `procEntryExit2()` and `procEntryExit3()` can be found within Appendix C.

4.3.3 Optimizations

The Concurrent MiniJava compiler contains a small number of arithmetic optimizations to increase the speed of certain calculations. In the ARM instruction set, most opcodes do not require more than a single clock cycle to complete [19]. However, it is a desirable educational opportunity for students to see how optimizations can be made to improve the performance of a compiler's output.

In `Codegen.java`, the code generator for each architecture, binary operations are associated with their assembly opcode counterparts. When certain arithmetic operations are being compiled, a check is done to see if the operand of either a multiply or divide is a power of 2. If this is true, in the case of a multiplication operation, the operand is shifted left the number of bits as its power of two. For example, if we want to multiply 8 by 2, our compiler will shift 8 left one bit rather than using the `mul` opcode. The shift algorithm is given below:

```
private static int shift(int i) {
    int shift = 0;
    if ((i >= 2) && ((i & (i - 1)) == 0)) {
        while (i > 1) {
            shift += 1;
            i >>= 1;
        }
    }
```

```
        return shift;
    }
```

For architectures that do not implement a multiply or divide opcode, repeated addition or subtraction could be used to achieve those operations. In that case, this sort of bit shifting optimization would increase the speed of that particular program since looping via branching is more expensive.

4.4 Educational Opportunities for Compilers and Security Students

Given this overview of our effort to retarget the Concurrent MiniJava compiler from the MIPS architecture to the ARM architecture, we must ask, "what does this achieve?" We know from our review of the literature, compiler construction is a rather difficult subject for students to grasp in a mere one semester course. As we will discuss further in Chapter 6, Marquette University students studying computing-related fields will learn about operating systems by implementing portions of the Embedded Xinu kernel. We choose to port Concurrent MiniJava to ARM because the latest version of Embedded Xinu runs on ARM-based Raspberry Pi 3B+ computers. It is a worthwhile exercise to target this platform because it is rich with classroom and research opportunities. At Marquette University, Embedded Xinu has been used to teach operating systems [20], elements of computer security [14], elements of networking [21], and previous iterations of the compiler construction course [2].

For students recently completing other courses where ARM Embedded Xinu is used, moving into the MiniJava environment can be easier than if the compiler was still the MIPS version. Carrying previous knowledge from course to course will help students retain new information. Writing a compiler is largely an exercise in pattern matching. A compiler accepts a program in text form, recognizes patterns in characters, and applies those patterns to a grammar. It takes those structures recognized by the grammar and constructs a tree encapsulating the semantics of the original program. It manipulates that tree and uses it to output another piece of text with equivalent meaning. If a student is already familiar with how system calls are

implemented in Embedded Xinu, understanding the MiniJava API and ARM calling conventions will be easier than learning a new architecture altogether.

Similarly, the exposure to a familiar assembly language helps students learning about decompiling binaries to analyze their vulnerabilities. A common exercise for students learning about computer security is to examine a decompiled executable to attempt to work out the function of the source code. A computer security instructor could use the Concurrent MiniJava compiler to provide to students ARM assembly code and the MiniJava API and ask them to analyze the program to figure out its function. More advanced courses might ask students to write MiniJava programs to attempt to uncover vulnerabilities within the Xinu kernel.

CHAPTER 5
Extending the Grammar With Access Modifiers

5.1 Access Modifiers in Java

Access modifiers are useful tools in helping to enforce encapsulation within object-oriented languages. In the Java language, access modifiers take the form of `public`, `private`, and `protected` "modes", denoted with the aforementioned keywords. Java also has the notion of the "default" access modifier. Unlike `public`, `private`, and `protected`, there is no discrete keyword to indicate that a variable, method, or object falls under the default access. When declaring one of these language structures, the programmer need only omit an access modifier to assign that structure the default access mode. Consider the following Java program:

Figure 5.1 contains two classes, `A` and `B`. Class `B` contains three private integer fields, `x`, `y`, and `z`. These fields are instantiated within `B`'s constructor. Within the main method of class `A`, a new object of type `B` is declared. On the last line of the main method, we intend to access the `x` field of `B` and print it to the console. However, when we attempt to compile this program, the compiler throws a syntax error:

```
error: x has private access in B
            System.out.println(objB.x);
                                   ^
```

This shows us an example of *encapsulation* within the Java language. Fields `x`, `y`, and `z` are encapsulated within and are not meant to be accessed outside of class `B`. The same idea can be applied to Java methods.

In Figure 5.2, classes `C` and `D` are defined. Class `D` contains two integer fields, `i` and `j`, with default access. Public method `foo` and private method `bar` are also defined and are called within the main method of class `C`. Much like the example shown in Figure 5.1, the call of method `bar` will cause a compile-time error. However, the call of method `foo` would succeed in executing if not for the error on the line

```
1   class A {
2       public static void main(String[] args) {
3           B objB = new B();
4           /* The line below will cause a compile-time error */
5           System.out.println(objB.x);
6       }
7   }
8   class B {
9       private int x;
10      private int y;
11      private int z;
12
13      public B() {
14          this.x = 5;
15          this.y = 6;
16          this.z = 7;
17      }
18  }
```

Figure 5.1: Illegal private field access

```
1       class C {
2           public static void main(String[] args) {
3               D objD = new D();
4               System.out.println(objD.foo());
5               System.out.println(objD.bar());
6           }
7       }
8
9       class D {
10          int i;
11          int j;
12
13          public D(int i, int j) {
14              this.i = i;
15              this.j = j;
16
17              this.j = this.bar();
18          }
19
20          public int foo() {
21              return this.i;
22          }
23
24          private int bar() {
25              return this.i * this.j;
26          }
27      }
```

Figure 5.2: Public and private method access

below. Private methods are meant to be called only within the class in which they have been defined. Private fields, on the other hand, can be accessed outside of their classes through the use of *setter* and *getter* methods. These methods would be given public access and would set and return the values of private fields. In this way, the programmer can preserve encapsulation and, thus, the security of the program as a whole.

Declaring all fields and methods with public access is not always a desirable design decision. When classes become components of larger systems, it becomes necessary to create a design where information can be kept hidden from outside modules and outside entities. This feature allows the programmer to write secure object-oriented code.

5.2 Adding Encapsulation to MiniJava

Adding encapsulation in the form of access modifiers to the Concurrent MiniJava language is done within both the Parsing and Type Checking phases of the compiler. We have modified the Concurrent MiniJava grammar to consider `public` and `private` as reserved words. Additionally, we have modified the notion of a base type. Within the Concurrent MiniJava grammar, a type can be either a boolean, an integer, an identifier (for a class/object), or the array variant of either of these. A base type is the non-array variant of a type. We made this design decision in order to associate the notion of an access modifier with the primitive variable or object itself, rather than distinguish between public/private arrays and public/private integers, booleans, or objects. In addition to public and private variables, Concurrent MiniJava now supports public and private methods. In a similar way to that of modifying the grammar to keep track of an access modifier flag for base types, a access modifier flag has been added to the abstract syntax tree classes for method declarations (`MethodDecl`) and void declarations (`VoidDecl`).

To modify the grammar, we introduce the notion of an AccessType. Figure 5.3 shows a subset of the Concurrent MiniJava grammar that defines an AccessType.

```
      <Type>  ::=  <BaseType> ( "[" "]" )*
<AccessType>  ::=  "public" | "private"
  <BaseType>  ::=  ( <AccessType> )?  <BaseType>
                |  "boolean"
                |  "int"
```

Figure 5.3: AccessType Grammar Rules

An AccessType is an optional (zero or one) non-terminal that comes before the declaration of a BaseType. This is an optional non-terminal. Examples of valid Type declarations include:

```
public int foo;          public boolean oof;
private int bar;         private boolean rab;
int baz;                 boolean zab;
```

Method declarations are modified in a similar fashion. The grammar distinguishes between a MethodDeclaration and a VoidDeclaration. MethodDeclarations require a return type and a return expression. VoidDeclarations do not have these requirements. To add access modifiers to these non-terminals, their production rules are changed by simply adding an optional AccessType to the beginning of each production:

```
<MethodDeclaration> ::= ( <AccessType> )? ( "synchronized" )? <Type>
                        "(" ( <Type>  ( "," <Type>  )* )? ")"
                        "{" ( <VarDeclaration> )*
                         ( <Statement> )* "return" <Expression> ";" "}"

  <VoidDeclaration> ::= ( <AccessType> )? "void"  "(" ")" "{"
                          ( <VarDeclaration> )* ( <Statement> )* "}"
```

In order to keep track of whether an identifier or method is public or private, we implement an additional constructor for each of the following abstract syntax tree classes: `MethodDecl`, `VoidDecl`, `IntegerType`, `BooleanType`, and `IdentifierType`. The abstract class `Type` gains an additional field, `isPrivate`, that `IntegerType`,

`BooleanType`, and `IdentifierType` inherit. This simple flag, included in the overloaded constructor, stores the access modifier status that gets checked during the type checking phase of the compiler.

The Type classes to be used during type checking have also been modified to include an `isPrivate` field. This allows the symbol table to store the access modifier status for each identifier and method name. When method and field access expressions are type checked, the value of `isPrivate` is also checked. If a private field is trying to be accessed outside of its class, the type checker raises an error and halts the remainder of the compiler's pipeline execution.

Consider a simple MiniJava program. Given two classes, `A` and `B`, an object of type `B` is declared within the main method of class `A`. Suppose `B` objects contain a private integer field `x`. If class `A` contains a statement that tries to access `x`, the following error would be raised by the compiler:

```
ERROR cannot access private field ‘x‘ outside of OBJECT:B:
          FieldExpr(
           NewObjectExpr(IdentifierType(B))
           x)
```

A similar error message is raised if a private method tries to be called outside of its class. Suppose class `B` has a private method called `getX()`. If `getX()` is called on `B` inside of class `A`, the following error would be raised:

```
ERROR cannot call private method getX outside of class OBJECT:B:
          CallExpr(
           NewObjectExpr(IdentifierType(B))
           getX
           AbstractList())
```

Access modifiers are a simple language feature that adds built-in protections to the language. For the MiniJava use case, they can be an instructive tool for both beginner and advanced students alike. For less-experienced students, the smaller

language set that MiniJava offers can be less intimidating. The notion of a public or private variable can be seen in a smaller environment than an example written in Java. For more experienced students, like graduate students, implementing access modifiers into a compiler can be a useful way to test a student's understanding of this language feature and can show a student how information moves throughout the compiler pipeline.

CHAPTER 6
Integrating with Embedded Xinu

6.1 Porting Concurrent MiniJava Built-In Functions

The previous effort to add MiniJava support to the Embedded Xinu kernel was completed in 2010 by Adam Mallen [2]. Since that time, the Embedded Xinu kernel has been ported to new ARM-based platforms [15]. With that effort comes the challenge of refreshing decade-old code and integrating it with an incompatible "modern" Xinu kernel. Thankfully, the effort to revamp the old MiniJava API to the most current version of Xinu was not insurmountable. The algorithms previously written to allocate new objects, to create new MiniJava threads, and to lock and unlock monitors could be rewritten largely identically.

The largest difference between the two versions of Embedded Xinu was that the legacy version, running on MIPS-based Linksys routers, was not a multicore operating system. The Linksys routers used single core processors, so that version of Xinu had no notion of multi-core scheduling. In order to allow MiniJava to achieve real concurrent programming capabilities, the procedure for creating new MiniJava threads had to be modified to include the processor core on which that thread would run.

Figure 6.1 shows the updated `_threadCreate` MiniJava API function supporting multi-core threading. We introduce a naïve round-robin method to spawn each new thread on a separate core. Using a static integer `nextcore`, we start the first thread on core 0, which contains the main MiniJava program's process. Each successive thread that gets created in MiniJava gets started on the next core, in order, until all four cores contain a thread. Then, we wrap around to core 0 and start the process over. Without this round-robin mechanism in thread creation, each thread would run on the same core as the main program. Threads executing synchronized methods would not experience true concurrency and Xinu's notion of monitors would not correctly enforce mutual exclusion among running threads. In fact, concurrently

```
1  syscall _threadCreate(int *threadObjAdder)
2  {
3      // back up one word to point to vtable pointer
4      int *A = threadObjAdder - 1;
5      // point to first entry of vtable
6      int *B = (int *)*A;
7      // point to first method's actual code
8      int *C = (int *)*B;
9      void *procadder = (void *)C;
10     static int nextcore = 0;
11     nextcore = (nextcore + 1) % 4;
12     return
13         ready(create
14               (procadder, INITSTK, INITPRIO, "MiniJavaThread", 1,
15                threadObjAdder), RESCHED_NO, nextcore);
16 }
```

Figure 6.1: Round-robin Thread Creation in Embedded Xinu

running threads would not exist at all.

Mallen's monitor module from MIPS Xinu needed some work to make it compatible with ARM Xinu. Monitors are the tool by which Java threads can operate with mutual exclusion. Monitors are similar to binary semaphores in that they can be locked to prevent other resources attempting to control it. The Xinu implementation of a monitor uses a semaphore for this purpose. A monitor is also able to keep track of the number of times it has been locked by its owning thread. This is useful because it determines whether a monitor is owned or unowned. If it is unowned, it can be locked by another thread. There are a fixed number of monitors allocated by Embedded Xinu. Since the previous monitor implementation did not have knowledge of more than one core, the monitor lock and unlocking operations had to call upon the table of currently running processes for its owning core to determine which process would own a particular monitor.

Figure 6.2 shows the monitor locking action found in Embedded Xinu's `lock.c`. If a monitor has not been allocated, its owner process is set to `NOOWNER`. The owner becomes the currently running process on that process' core. To show that the monitor had a locking action performed, its counter is incremented and then the process

```
1  monptr = &montab[mon];
2
3  /* if no thread owns the lock, the current thread claims it */
4  if (NOOWNER == monptr->owner)
5  {
6      /* current thread now owns the lock  */
7      monptr->owner = currpid[cpuid];
8      /* add 1 "lock" to the monitor's count */
9      (monptr->count)++;
10     /* this thread owns the semaphore */
11     wait(monptr->sem);
12 }
13 else
14 {
15     /* if current thread owns the lock increase count; dont wait
          on sem */
16     if (currpid[cpuid] == monptr->owner)
17     {
18         (monptr->count)++;
19     }
20     /* if another thread owns the lock, wait on sem until monitor
           is free */
21     else
22     {
23         wait(monptr->sem);
24         monptr->owner = currpid[cpuid];
25         (monptr->count)++;
26     }
27 }
```

Figure 6.2: Locking a Monitor

waits on the monitor's semaphore. Otherwise, if the monitor is already owned by a process, we just increment the counter; it is not necessary to wait. If we want to lock another thread's monitor, we must wait until the monitor's semaphore allows us to acquire it.

Unlocking a monitor is a simpler. We can know if a monitor is locked because its counter will be greater than zero. If we unlock a monitor, we decrement its counter by one. If, after this decrement, the monitor's counter becomes zero, we reset its owner to the `NOOWNER` macro and signal its semaphore to notify another process potentially trying to access it.

This lock/unlock action enforces mutual exclusion among threads. Since MiniJava is a subset of Java, each construct in the language is itself an object. Therefore, at the time of creation, each object gets allocated its own monitor. For a full code listing of the MiniJava API, including creating a new object, see Appendix B.

6.2 Implementing `_lock` and `_unlock` System Calls During Translation

In order to successfully enforce method synchronization among threads, we must use the lock and unlock functionality described above. This occurs during the translation phase of the compiler. The translation phase consists of mapping type-decorated abstract syntax tree notes to instructions in the compiler's intermediate representation. Within the act of translating a method declaration (`MethodDecl`), we must check to see if that method is synchronized. Synchronized methods contain a boolean attribute `synced`. If a method's `synced` attribute is true, we wrap that method boy with an external call to `_lock()` and _unlock(). To do this, we add a call to `_lock()` into the beginning of the method's body and pass `_lock()` the `this` pointer of the method's parent class. See Figure 6.3 for the implementation of this action. The lock function in the MiniJava API takes a pointer to an object as its argument. The `this` pointer serves the role of pointing to the object that contains the implementation of the synchronized method. The MiniJava API then locks the monitor associated with the object.

```
1  if (m.synced) {
2      Absyn.ThisExpr t = new Absyn.ThisExpr();
3      args.add(t.accept(this).unEx());
4
5      body = SEQ(body, EXP(frame.externalCall(Symbol.get("lock"),
           args)));
6  }
```

Figure 6.3: Appending a Call to `_lock()` to the Body of a Synchronized `MethodDecl`

```
1  if (m.synced) {
2      Temp tmp = new Temp();
3
4      /* evaluate return expression and move to temporary location
           */
5      Exp rv = null;
6      rv = m.returnVal.accept(this);
7
8      /* Exp tmpmove = new Nx(new Tree.MOVE(new Tree.TEMP(tmp), rv.
           unEx())); */
9      body = SEQ(body, MOVE(TEMP(tmp), rv.unEx()));
10
11     /* do unlock */
12
13     body = SEQ(body, EXP(frame.externalCall(Symbol.get("unlock"),
           args)));
14
15     /* move return expression back into R0 */
16     rv = new Nx(new Tree.MOVE(new Tree.TEMP(frame.RV()), new Tree
           .TEMP(tmp)));
17
18     body = SEQ(body, (MOVE(TEMP(frame.RV()), TEMP(tmp))));
19 }
```

Figure 6.4: Appending a Call to `_unlock()` to the Body of a Synchronized `MethodDecl`

Inserting a call to `_unlock()` is not as simple as what we see above. We must place the call to `_unlock()` after the return statement of the method. This creates a problem, however. If we evaluate the return statement, we leave the method entirely and never perform the unlocking action. In order to get around this problem, we evaluate the return expression and move it into a temporary location within the stack frame. Then, we perform the unlock. Afterwards, we move the return expression back and perform the return. See Figure 6.4 for the implementation of this action.

6.3 Educational Opportunities for Operating Systems Students

At Marquette University, the modus oprerandi for teaching Operating Systems is to use the Embedded Xinu platform. Our work brings Concurrent MiniJava to the latest version of the Embedded Xinu kernel. It fosters learning with contiguous pedagogical tools. Students at Marquette will build critical components of the Xinu kernel throughout their one semester Operating Systems class. The skills acquired in that class, including familiarity with Embedded Xinu, will carry over to Compiler Construction and even courses that utilize networking and security concepts.

While learning how to implement an operating system, a student will inevitably come across the necessity to implement critical sections of a kernel in assembly language. There are some parts of the Embedded Xinu kernel, like the context switch, that must directly touch the hardware. If a student can engage with assembly language in more than one course, they will attain a greater degree of familiarity with how programs work at such a low level. Using a version of Embedded Xinu that supports the MiniJava API and monitors, students can see exactly how a program works, from existing only in a text file, to assembly language, to creating a process, and finally to executing instructions on bare metal.

CHAPTER 7
Challenges Overcome

Throughout the course of this research, we have encountered roadblocks that were challenging to overcome. In this chapter, we will outline two main issues and explain how they were resolved.

7.1 Memory Offsets with the Stack and Frame Pointers

ARM stack frames use both the stack pointer and frame pointer to denote each end of the frame. In industrial practice, the frame pointer stays fixed at the base of the stack frame and the stack pointer can move as the frame grows down. Using this technique, data in memory addressed by offsets from an area in the frame can use the frame pointer as their base. Each offset would be some distance away from the frame pointer. This is a good choice since the stack pointer is not guaranteed to stay in the same location throughout the life of the frame. The Concurrent MiniJava compiler for both MIPS and ARM uses a different technique. Instead of having a "floating" stack pointer, the relative address of the stack pointer is decided at compile time. When a stack frame is being set up during code generation, the size of the stack frame is calculated and stored as a macro, from which each offset address is calculated.

Consider the code snippet in Figure 7.1, containing the beginning of a simple MiniJava program. The `main` method of this class prints "Hello world from MiniJava!" ten times using a while loop. We can see that the compiler has calculated the size of this stack frame and included it as a macro called `main_framesize`. The values contained within `r4` and `r5` are being stored into the memory address found within the stack pointer register, offset by values calculated with `main_framesize`. During development of the ARM backend, we decided to try using the frame pointer as the base upon which offsets would be calculated. While this compiler was designed to be extensible to other architecture. the MIPS version was written with design decisions favoring MIPS. The MIPS backend uses the stack pointer for offsets and switching to using the frame pointer for other architectures was not favorable and caused issues

```
1  #include <arm.h>
2          .section .data
3  L0:     .asciz  "Hello␣world␣from␣MiniJava!"
4          .section .text
5          .align  4
6          .globl  main
7  main:
8  #define main_framesize 16
9          push    { fp, lr }
10         add     fp,     sp,     #4
11         sub     sp,     sp,     #main_framesize // create stack
               frame
12         str     r4,     [sp, #-8+main_framesize]
13         str     r5,     [sp, #-4+main_framesize]
14         mov     r5,     r6
```

Figure 7.1: Frame Size Offsets

as ARM functionality was increased. We use the stack pointer as our base for offset calculation because this MiniJava compiler does not require the stack pointer to be shifted elsewhere within a stack frame. In a language that does require this, using the stack pointer would not be an appropriate choice for offsets.

7.2 Assembler Errors with Large Constants

The ARMv7 instruction set can only support operations on immediate values that are smaller than 16 bits. Certain test programs used for debugging during the development of the ARM back-end contain integer constants that are larger than 16 bits. Recall that the largest 2's complement signed integer that can be represented in 16 bits is $2^{15} - 1$, or $32,767$. The MIPS implementation of the Concurrent MiniJava compiler contained a function to check if a constant is 16 bits in size. This method, `CONST16()`, found within `Codegen.java`, can be seen below. As a clarifying note, the size of a Java `short` is 16 bits.

```
private static boolean CONST16(Tree.CONST c) {
    return c.value == (short)c.value;
}
```

This method was used in the MIPS implementation of the compiler because MIPS32 has the same constraint as ARMv7 regarding 16 bit constants. There are

multiple solutions to this problem. The first is slightly complex. One could split a large constant, 70,000, for example into an upper and lower half. In signed 2's complement binary, 70,000 is 00000000 00000001 00010001 01110000. This 32 bit number, separated into four bytes for ease of reading, can be split into two two-byte parts: 00000000 00000001 and 00010001 01110000. Assuming we are working in a big-endian architecture, the "most significant" half is 00000000 00000001 and the "least significant" half is 00010001 01110000. If we pad the lower half of the most significant chunk up to 32 bits, we can load that into a register and perform a bitwise OR with the least significant chunk to "rebuild" our full number inside of a register.

A second, perhaps more elegant, solution is to simply store the full number as a 32 bit word in memory. We could append a `.data` section to the bottom of our program and store the word there. We could then reference that word's address in memory through some offset of the either the stack pointer or frame pointer. This might be an interesting exercise for a student to implement in the context of an advanced compiler construction course. Of course, however, we wish to **simplify** this endeavour for students learning about compilers for the first time. In order to aid in the learning process, our third solution is the one that is implemented within the ARM Concurrent MiniJava compiler.

The third solution is to let the assembler and linker do the work for us. The ARMv7 instruction set contains a useful pseudo-instruction to abstract-away the same process described in our second solution. We can use the `ldr` with the full *text* of the constant prefixed with an equals sign. For example, if we wanted to load register `r4` with the constant value 70,000, we would write:

```
ldr r4, =70000
```

When an ARM assembler sees this syntax, it, along with a linker, implements object code that creates a word, stores 70,000 at its address, and references that word from an offset. In `Codegen.java`, we can see this solution implemented in the `visit`

```
1  Temp d0 = new Temp();
2      if (CONST16(e)) {
3          emit(OPER("mov\t`d0,\t" + "#" + e.value, new Temp[]{d0},
               null));
4          return d0;
5      }
6      else {
7          emit(OPER("ldr\t`d0,\t" + "=" + e.value, new Temp[]{d0},
               null));
8          return d0;
9      }
10 }
```

Figure 7.2: Implementing Pseudo-Instructions for Constants Larger than 16 Bits.

method for the `CONST` IR node, seen in Figure 7.2. In this case, we use `CONST16()` to check if our constant `e` is 16 bits. If the constant is not 16 bits we know it must be larger, so we use the pseudo-instruction syntax. We do not need to check if the constant is less that 16 bits because it will be padded up to 16 bits in length.

CHAPTER 8
Conclusion

8.1 Summary of Contributions

In this thesis we have presented the results of our research in improving the Concurrent MiniJava compiler for the sake of compiler construction education. Our version of this compiler not only supports the latest version of the Embedded Xinu operating system for the Raspberry Pi 3B+, providing educational opportunities for students learning about compilers, operating systems, and computer security. The MiniJava language has an extended grammar to support access modifiers, providing additional learning opportunities for students taking coursework in programming languages. It is our hope that future students in several areas will benefit from this research effort and excel in their studies with an increased interest in compilers and gain an appreciation for the many years of fundamental research that went into making work like this possible.

8.2 Further Work

The most exciting and imminent application of this research is due to occur during the Fall 2022 offering of COSC 4400/5400: Compiler Construction at Marquette University. This latest iteration of the Concurrent MiniJava compiler will be used as an instructional tool for the first time, hopefully delivering the results we predict it will.

Natural extensions of this work include retargeting this compiler to additional platforms. While the Embedded Xinu operating system runs on a handful of platforms, it may be a worthwhile effort to consider implementing support for Linux systems. The easiest and most efficient use of resources would be to target a version of Linux that can be supported by a Raspberry Pi 3B+, like Raspbian OS. If the MiniJava API could be extended to support commonly used Linux system calls, MiniJava programs could be executed independently of Embedded Xinu.

An improvement can be made to the ARM backend of the compiler by intro-

ducing `PUSH` and `POP` nodes into the intermediate representation tree. As discussed in Chapter 4, the current ARM implementation does not push most volatile registers onto the stack via the `push` and `pop` opcodes. This avenue was explored, but ultimately was not implemented due to the lack of robustness in the intermediate representation. Refactoring the intermediate representation to include these additions would add more support to languages that make heavy use of the stack and increase the potential for stack-based language support.

REFERENCES

[1] A. W. Appel and J. Palsberg, *Modern Compiler Implementation in Java, Second Edition*. Cambridge University Press, 2002.

[2] A. B. Mallen and D. Brylow, "Compiler construction with a dash of concurrency and an embedded twist," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 161–168. [Online]. Available: https://doi.org/10.1145/1869542.1869568

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, & Tools, Second Edition*. Addison Wesley, 2006.

[4] JavaCC Community, "JavaCC: Introduction." [Online]. Available: https://javacc.github.io/javacc

[5] N. Chomsky, *Syntactic Structures*. Mounton & Co., 1957.

[6] S. C. Johnson, "Yacc: Yet another compiler-compiler." [Online]. Available: http://dinosaur.compilertools.net/yacc/

[7] M. L. Corliss and E. C. Lewis, "Bantam: A customizable, java-based, classroom compiler," *SIGCSE Bull.*, vol. 40, no. 1, p. 38–42, mar 2008. [Online]. Available: https://doi.org/10.1145/1352322.1352153

[8] M. L. Corliss, D. Furcy, J. Davis, and L. Pietraszek, "Bantam java compiler project: Experiences and extensions," *J. Comput. Sci. Coll.*, vol. 25, no. 6, p. 159–166, jun 2010.

[9] N. Wang and L. Li, "A hybrid teaching reform scheme of compiler technology course based on engineering education," in *Proceedings of the 5th International Conference on Information and Education Innovations*, ser. ICIEI 2020. New

York, NY, USA: Association for Computing Machinery, 2020, p. 16–19. [Online]. Available: https://doi.org/10.1145/3411681.3411695

[10] J. G. Politz and Y. Alhessi, "Course experience report: Full-class compiler collaboration," in *Proceedings of the 2021 ACM SIGPLAN International Symposium on SPLASH-E*, ser. SPLASH-E 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 19–25. [Online]. Available: https://doi.org/10.1145/3484272.3484961

[11] J. H. Lasseter, "The interpreter in an undergraduate compilers course," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 168–173. [Online]. Available: https://doi.org/10.1145/2676723.2677314

[12] D. Baldwin, "A compiler for teaching about compilers," in *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 220–223. [Online]. Available: https://doi.org/10.1145/611892.611974

[13] D. Comer, *Essentials of Computer Architecture Second Edition.* CRC Press, 2017.

[14] P. J. McGee, "An Educational Operating System Supporting Computer Security," Master's thesis, Marquette University, 2020.

[15] Marquette University Systems Lab, "List of supported platforms," 2013. [Online]. Available: https://xinu.cs.mu.edu/index.php/List_of_supported_platforms

[16] Raspberry Pi Foundation, "Raspberry pi 3 model b+ product brief." [Online]. Available: https://datasheets.raspberrypi.com/rpi3/raspberry-pi-3-b-plus-product-brief.pdf

[17] Arm Limited, "Arm Compiler armclang Reference Guide Version 6.6.2." [Online]. Available: https://developer.arm.com/documentation/dui0774/i/armclang-Integrated-Assembler-Directives/AArch32-Target-selection-directives

[18] MIPS Technologies Inc., *MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS32® Architecture, Revision 6.01.* MIPS Technologies Inc., 2014. [Online]. Available: https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00082-2B-MIPS32INT-AFP-06.01.pdf

[19] M. McDermott. (2008) The ARM Instruction Set Architecture. [Online]. Available: https://users.ece.utexas.edu/~valvano/EE345M/Arm_EE382N_4.pdf

[20] P. J. McGee, R. Latinovich, and D. Brylow, "Using Embedded Xinu and the Raspberry Pi 3 to Teach Operating Systems," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* Los Alamitos, CA, USA: IEEE Computer Society, may 2020, pp. 307–315. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IPDPSW50202.2020.00063

[21] D. Brylow, "COSC 3300 Networks and Internets: Fall 2011." [Online]. Available: https://www.cs.mu.edu/~brylow/cosc3300/Fall2011/

Appendices

Appendix A: Full Code Listing of `helloworld.c`

```c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");

    return 0;
}
```

Appendix B: Full Code Listing of minijava.c

```
1  /* Embedded Xinu, Copyright (C) 2008.  All rights reserved. */
2
3  #include <xinu.h>
4  #include <stdio.h>
5  #include <stdarg.h>
6  #include <stdlib.h>
7  #include <device.h>
8  #include <tty.h>
9  #include <memory.h>
10 #include <proc.h>
11 #include <monitor.h>
12 #include <yield.h>
13
14 syscall _readint(void)
15 {
16     int i = 0, c = 0;
17     c = kgetc((int)&devtab[SERIAL0]);
18     while (('\n' != c) && ('\r' != c) && (EOF != c))
19     {
20         if (('0' <= c) && ('9' >= c))
21         {
22             i = i * 10 + c - '0';
23         }
24
25         c = kgetc((int)&devtab[SERIAL0]);
26     }
27     kprintf("\r\n");
28     if (EOF == c)
29         return c;
30
31     return i;
32 }
33
34 syscall _printint(int i)
35 {
36     return kprintf("%d\r\n", i);
37 }
38
39 syscall _print(char *s)
40 {
41     return kprintf("%s", s);
42 }
43
44 syscall _println(void)
45 {
46     return kprintf("\r\n");
47 }
48
49 syscall _yield(void)
50 {
51     return yield();
52 }
```

```
53
54  syscall _sleep(int time)
55  {
56          return sleep(time);
57  }
58
59  int *_new(int n, int init)
60  {
61      int size = (n + 2) * 4;
62      int *p = (int *)getmem(size);
63      bzero(p, size);
64      p[0] = moncreate();
65      p[1] = init;
66      return p + 2;
67  }
68
69
70  syscall _lock(int *objAdder)
71  {
72      int *A = objAdder - 2;
73      monitor m = (monitor) * A;
74      return lock(m);
75  }
76
77  syscall _unlock(int *objAdder)
78  {
79      int *A = objAdder - 2;
80      monitor m = (monitor) * A;
81      return unlock(m);
82  }
83
84  syscall _threadCreate(int *threadObjAdder)
85  {
86      int *A = threadObjAdder - 1;
87      int *B = (int *)*A;
88      int *C = (int *)*B;
89      void *procadder = (void *)C;
90      static int nextcore = 0;
91      nextcore = (nextcore + 1) % 4;
92      return
93          ready(create
94                (procadder, INITSTK, INITPRIO, "MiniJavaThread", 1,
95                 threadObjAdder), RESCHED_NO, nextcore);
96  }
97
98  void _BADPTR(void)
99  {
100     fprintf(CONSOLE, "FATAL ERROR: Null Pointer Exception!\n");
101     unsigned int cpuid;
102     cpuid = getcpuid();
103     kill(currpid[cpuid]);
104 }
105
106 void _BADSUB(void)
```

```
107 {
108     fprintf(CONSOLE, "FATAL␣ERROR:␣Index␣Out␣Of␣Bounds␣Exception
        !\n");
109     unsigned int cpuid;
110     cpuid = getcpuid();
111     kill(currpid[cpuid]);
112 }
```

Appendix C: Code Listing of procEntryExit1(), procEntryExit2(), and procEntryExit3() in ArmFrame

```
1       public void procEntryExit1(List<Tree.Stm> body) {
2           assignFormals(formals.iterator(), actuals.iterator(),
                body);
3           assignCallees(0, body);
4       }
5
6       public void procEntryExit2(List<Assem.Instr> body) {
7           body.add(new Assem.OPER("\t//␣return␣from␣" + name, null,
                returnSink, null));
8       }
9
10      public void procEntryExit3(List<Assem.Instr> body) {
11          // for link regiter and frame pointer being pushed to the
                top of the stack at the top of a frame
12          int frameSize = maxArgOffset - localsOffset + 8;
13          ListIterator<Assem.Instr> cursor = body.listIterator();
14
15          // setup section info for a procedure
16          cursor.add(new Assem.OPER("\t.section␣.text", null, null,
                null));
17          cursor.add(new Assem.OPER("\t.align\t4", null, null, null
                ));
18          cursor.add(new Assem.OPER("\t.globl\t" + name, null, null
                , null));
19          cursor.add(new Assem.OPER(name + ":", null, null, null));
20          // setup a macro for the size of the stack frame to be
                created
21          cursor.add(new Assem.OPER("#define␣" + name + "_framesize
                ␣" + frameSize, null, null, null));
22
23          if (frameSize != 0) {
24                  // create stack frame
25                  cursor.add(new Assem.OPER("\tpush\t{␣fp,␣lr␣}",
                        null, null, null));
26
27                  cursor.add(new Assem.OPER("\tadd\tfp,\tsp,\t#4",
                        null, null, null));
28
29                  cursor.add(new Assem.OPER("\tsub\tsp,\tsp,\t"
30                                          + "#" + name + "
                                                _framesize\t\t//␣
                                                create␣stack␣
                                                frame",
31                                          new Temp[]{SP}, new
                                                Temp[]{SP}, null)
                                                );
32
33                  body.add(new Assem.OPER("\tsub\tsp,\tfp,\t#4",
                        null, null, null));
34
35                  body.add(new Assem.OPER("\tpop\t{␣fp,␣lr␣}", null
```

```
                        , null, null));
36          }
37
38          body.add(new Assem.OPER("\tbx\tlr", null, new Temp[]{LR},
                null));
39      }
```