JITED: A FRAMEWORK FOR JIT EDUCATION IN THE CLASSROOM

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Caleb Watts

December 2021
COMMITTEE MEMBERSHIP

TITLE: JITed: A framework for JIT education in the classroom

AUTHOR: Caleb Watts

DATE SUBMITTED: December 2021

COMMITTEE CHAIR: Aaron Keen, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Phillip Nico, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Ayaan Kazerouni, Ph.D.
Professor of Computer Science
The study of programming languages is a rich field within computer science, incorporating both the abstract theoretical portions of computer science and the platform specific details. Topics studied in programming languages, chiefly compilers or interpreters, are permanent fixtures in programming that students will interact with throughout their career. These systems are, however, considerably complicated, as they must cover a wide range of functionality in order to enable languages to be created and run. The process of educating students thus requires that the demanding workload of creating one of the systems be balanced against the time and resources present in a university classroom setting. Systems building upon these fundamental systems can become out of reach when the number of preceding concepts and thus classes are taken into account. Among these is the study of just-in-time (JIT) compilers, which marry the processes of interpreters and compilers for the purposes of a flexible and fast runtime.

The purpose of this thesis is to present JITed, a framework within which JIT compilers can be developed with a time commitment and workload befitting of a classroom setting, specifically one as short as ten weeks. A JIT compiler requires the development of both an interpreter and a compiler. This poses a problem, as classes teaching compilers and interpreters typically feature the construction of one of those systems as their term project. This makes the construction of both within the same time span as is usually allotted for a single system infeasible. To remedy this, JITed features a prebuilt interpreter, that provides the runtime environment necessary for the compiler portion of a JIT compiler to be built. JITed includes an interface for students
to provide both their own compiler and the functionality to determine which portions of code should be compiled. The framework allows for important concepts of both compilers in general and JIT compilers to be taught in a reasonable timeframe.
ACKNOWLEDGMENTS

Thanks to:

- Aaron Keen, for being an ever-patient font of knowledge throughout the process of making my thesis

- Ayaan Kazerouni and Phillip Nico, for being excellent teachers in my classes and being on my thesis committee

- Kiley Roberson, for working with me on a compiler that would form the blueprint for the example compiler featured here

- My family, for supporting me regardless of any difficulty or how long it took
# TABLE OF CONTENTS

| LIST OF TABLES                          | x  |
| LIST OF FIGURES                        | xi |

## CHAPTER

1  Introduction                         | 1  |
   1.1 Introduction                      | 1  |
      1.1.1 Motivation                   | 2  |
   1.2 Contributions                     | 3  |
      1.2.1 Proposed Learning Topics     | 4  |
   1.3 Requirements                      | 5  |
      1.3.1 Requirement 1: Identifying hot code | 5  |
      1.3.2 Requirement 2: Compiling LLVM intermediate representation | 6  |
      1.3.3 Requirement 3: Invoking compiled code | 6  |
      1.3.4 Requirement 4: Standalone interpreter | 7  |
      1.3.5 Relation to Coursework       | 7  |
   1.4 Organization                      | 7  |
   2  Related Work                       | 8  |
      2.1 Background                     | 8  |
      2.2 Related Work                   | 8  |
         2.2.1 JIT Optimizations          | 9  |
         2.2.2 JIT Register Allocation    | 10 |
         2.2.3 JIT Security               | 11 |
            2.2.3.1 Experiments with Timing Variation | 12 |
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Unpatched, secret-sensitive worker function across 1,000,000 total invocations. Time ratios closer to 1.00 indicate less susceptibility to timing attacks.</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Patched worker function across 1,000,000 total invocations. Time ratios closer to 1.00 indicate less susceptibility to timing attacks.</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>An example of functions being called and how many times they were, in order. The rightmost three columns list whether they would be JIT'd using the default, threshold, and proportion heat functions, respectively.</td>
<td>25</td>
</tr>
<tr>
<td>5.1</td>
<td>Description of the TypedValue class.</td>
<td>37</td>
</tr>
<tr>
<td>5.2</td>
<td>Description of the PackedStruct class.</td>
<td>40</td>
</tr>
<tr>
<td>5.3</td>
<td>Description of the MiniInterpreter class.</td>
<td>44</td>
</tr>
<tr>
<td>5.4</td>
<td>Description of the JIT class.</td>
<td>52</td>
</tr>
<tr>
<td>5.5</td>
<td>Description of the ModuleCompiler class.</td>
<td>56</td>
</tr>
<tr>
<td>6.1</td>
<td>Estimated timeline of the term project.</td>
<td>60</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 A diagram of the system’s components. Student-written components are bolded and have rounded corners.</td>
<td>14</td>
</tr>
<tr>
<td>4.1 Examples of heat functions.</td>
<td>24</td>
</tr>
<tr>
<td>4.2 A short example Mini program. The Mini code is in a box in the top left, while the tree on the right is the parsed AST. Edge labels indicate the field name, node labels indicate the object type.</td>
<td>26</td>
</tr>
<tr>
<td>4.3 An example of the kind of output that the JIT expects to be returned by a compiler.</td>
<td>28</td>
</tr>
<tr>
<td>4.4 The line in which the compiler is invoked from the JIT.</td>
<td>29</td>
</tr>
<tr>
<td>5.1 Potentially counterintuitive behavior: the line where f is defined throws an exception, whereas if there were not the intermediate usage of Any, it would work.</td>
<td>36</td>
</tr>
<tr>
<td>5.2 Functional upcasting of values contained within an antlrcpp::Any. The Any itself should be constructed using the proxy function. Extraction can then be done normally.</td>
<td>37</td>
</tr>
<tr>
<td>5.3 An example of how PackedStructs are linked internally. Code that generates this example is shown in the top left.</td>
<td>43</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

1.1 Introduction

Programming languages lie at the core of computer science. Even if the study of these languages themselves is not of particular interest to all students, they still interact with languages by using them. At least a basic understanding of programming languages is thus imperative. If students choose to pursue the field further, however, there is a rich array of topics to dive into. These span abstract concepts such as grammars and properties of languages, to implementation details and the specifics of architectures.

One of the main aspects of programming languages is in how a language is executed. Broadly, languages are either interpreted or compiled. These two approaches lend themselves well to university-level classes, both undergraduate and graduate. Generally, interpretation is presented as a way of dynamically running code, while compilation is a static way of running code, being an ahead of time process. This can be framed as an interpreter being a flexible, albeit slower way of running code. In contrast to this, compilers and the created executables are fast, but not inherently portable and can require setup in advance of actually executing the program.

Adjacent and parallel to these is the concept of just-in-time (JIT) compilation. JIT compilation merges the idea of beginning program execution with the source code as input, as happens with an interpreter, and some of the performance-oriented processes of compilers. JIT compilation is a system where code is initially interpreted, and then heavily used sections (often referred to as "hot" segments) of the program are compiled
into a form where it can be natively executed by the system. This is the system used by Java, a popular programming language. Systems such as HotSpot[19] implement a Java virtual machine (JVM) that provides the environment for JIT compilation.

While JIT compilation is a technology that is actively being used in industry, it involves a great number of different concepts from programming languages. It combines the complexity of both interpretation and compilation, along with unique challenges including the security concerns of creating executable sections of memory at runtime. This makes reasonable knowledge of how to construct both an interpreter and a compiler individually a prerequisite to creating a JIT-compiled system. This makes JIT compilation a difficult prospect to teach in a college environment, where the prerequisite classes of a course must be carefully considered. The goal of the thesis that this document is a part of is to provide a framework for students to work within that would allow exploration of JIT concepts without requiring as much prior knowledge as building one from scratch. The requirements of the platform were to support meaningful labs and assignments related to JIT compilers, without performing too much of the process for the students. The precise requirements set forth will be discussed later in this chapter.

1.1.1 Motivation

Our motivation is to facilitate the teaching of JIT concepts to students. The technology is a prevalent factor in modern programming languages, and is somewhat underrepresented in education despite this. Studying JIT compilers would introduce students to concepts within the scope of programming languages that they would otherwise not be exposed to, and emphasize aspects of compilation and interpretation that might not otherwise seem to have a practical impact. Examples of the former would include things like the process and benefits of profiling a program, the creation
of executables and the process of loading them into the operating system to run, and more. The latter includes topics like the impact of the compiler’s runtime, and the need to intelligently balance the quality of produced code against the computation required to create that code in the first place.

The topic of JIT compilation is also an additional point of engagement for students in programming languages. JIT compilation has a particular focus on real-world performance gains, and ties into the more mechanical aspects of the computer science discipline. This provides a more concrete focus for a course, involving less of the abstract concepts that would likely be present in prior programming languages courses. In turn, this could lead to greater engagement, or engagement with a distinct population of students from prior courses.

Additionally, JIT compilers present a larger attack surface from a security standpoint, as they necessarily dynamically write executable sections of memory. This aspect of JIT compilers is outside the purview of the framework presented in this thesis, but is an important concern in the field of JIT compilers. It will therefore be briefly discussed in this thesis, and we recommend that any coursework incorporating this thesis’ framework also include JIT security as a topic.

1.2 Contributions

The main contribution of this thesis is the production of a framework for handling some of the tasks associated with JIT compilation, named JITed. In essence, the framework is meant to handle the tasks like interpretation that are required for a JIT system to function, but are not unique to JIT systems and exist independently in other contexts. To wit, this includes the major components of the parser, the interpreter, and the compilation from intermediate representation to executable code. This leaves
to the student the tasks of compiling from source to intermediate representation, deciding when and how code should be compiled, and optimization tradeoffs involved in these.

The language that JITed is developed for is Mini[12], a minimally-featured language with C-style syntax created by Dr. Aaron Keen[13]. The language represents a base of features to support simple programming, with support for functions, structs, allocation and deallocation of dynamic memory, and basic IO. Further detail on Mini can be found in Chapter 3 and in the appendices.

1.2.1 Proposed Learning Topics

The following section lays out the general goals that the framework was developed to accommodate. These generally take the form of teaching some topic related to JIT compilers, or enabling students to develop a portion of a JIT compiler smoothly. These topics are put forth to frame the discussion of the framework features and the example labs later in the document. These topics are only those that can be executed within the framework itself; other topics, such as the topic of JIT security, should be carried out in other contexts.

The proposed topics are as follows:

- **T1**: The components of a JIT compiler.
- **T2**: Learning techniques of determining hot code.
- **T3**: Building components of a compiler.
- **T4**: Optimizing for workloads with a JIT compiler.
These topics are to be presented alongside other core compiler concepts, including but not limited to: control flow graphs, phis/coalescing, and optimization techniques in general.

1.3 Requirements

To frame our evaluation of the system, we set forth the following requirements. These are phrased in terms of what students should be able to do with the system, and directly relate to the JIT principles that the framework is intended to help teach.

- **REQ1**: Students should be able to identify and designate hot (i.e. frequently used) sections of code.
- **REQ2**: Students should be able to compile from minimal LLVM intermediate representation code.
- **REQ3**: Students should be able to run compiled code when desired.
- **REQ4**: The interpreter should function opaquely to the user, without direct access.

1.3.1 Requirement 1: Identifying hot code

The first requirement relates to the process of choosing which functions to JIT compile. The process of JIT compiling carries significant overhead; the code must be converted into assembly or some intermediate representation, optimized, and then compiled and linked into runnable memory segments. Students should therefore have the appropriate resources to make intelligent decisions about which functions to compile. These should be a part of the outwardly-facing JIT interface. The goal from
doing this is to allow students to experiment with different degrees of aggressiveness for JIT compilation, and different techniques for choosing functions to compile.

### 1.3.2 Requirement 2: Compiling LLVM intermediate representation

The second requirement is that the student should be able to submit, in string form, LLVM intermediate representation and have it be integrated into the runtime as runnable memory. This provides the student with a reasonable target language; LLVM IR is relatively similar to assembly languages, but also allows for virtual registers over physical ones, as well as typed values and other abstractions.

Furthermore, students should be able to compile a function and run the code without supplying more than the function definition. This is to say that the student will not be including the declarations or struct information that would otherwise be required to compile their LLVM IR code. Determining what information would be required for each individual function to properly compile is potentially nebulous to students, especially as they do not have full information of how the final compilation is being done.

### 1.3.3 Requirement 3: Invoking compiled code

The third requirement is simply a statement of how the system should work. The student’s first interactions with the system will likely be with the interpreter only, but they should be able to freely choose which code should be run in compiled form. This means that the student should be able to act upon knowing a function is hot, and switch it from being interpreted to running natively.
1.3.4 Requirement 4: Standalone interpreter

The fourth requirement is about what tasks should not be required of the student. Specifically, it is that the student should be able to develop their program without needing to explicitly interact with the interpreter. The required knowledge and effort should be constrained to writing a compiler, and not any component of an interpreter.

1.3.5 Relation to Coursework

In the evaluation, these requirements and topics will be revisited alongside proposed lab structures that both show that the requirement has been met, and that the concept it is intended to demonstrate is properly covered. Generally, this takes the form of presenting one of the requirements, then showing the features of the framework that enable it to be fulfilled.

1.4 Organization

This thesis document is arranged as follows. In Chapter 2, the related work and basic concepts relevant to the thesis are discussed. In Chapter 3, an overview of JITed’s system design is given. This covers the components of the system at a high level, along with the intent behind some of the design decisions made. In Chapter 4, details of the implementation of JITed are given. Listings of the classes used, how components relate to each other, and how instructors and students can interact with the system are given here. In Chapter 5, the system is evaluated in terms of its ability to meet the requirements stated. In Chapter 6, closing thoughts and conclusions are given, with the consideration of future work or adaptations discussed.
Chapter 2

RELATED WORK

2.1 Background

JITed is a framework intended to replicate a just-in-time (JIT) compiler, with student input. The full implication of this is that an interpreter is initially used in order to run the program from a parsed structure from the source code. During this interpreted phase, a lightweight profiler is used to gather statistics on the functions or code traces that account for most of the program’s runtime. This can be done with methods like tracking the number of calls to a function, or intermittently sampling the last called function, roughly estimating how much time is spent inside a function. Based on this, the code for frequently used code segments can be compiled and optimized.

The performance gain from JIT compilation lies both in the advantages inherent to running code natively and the ability to optimize the code. Native code both reduces the instructions required compared to using the interpreter and allows more effective use of registers over memory access. Optimizations, while typically fairly light, also yield a performance boon.

2.2 Related Work

The section that follows discusses the previous published work in the domain of JIT compilers and related topics which have influenced this thesis. Also discussed is an experiment run by the thesis’ author as part of the preceding work, which relates to JIT security. JIT security in general is an important subject and one that should be...
discussed with students. Hands-on labs dealing with JIT security should be held on a system that more closely matches a production environment, however.

### 2.2.1 JIT Optimizations

JIT compilation allows for optimizations to be performed on the code that would not otherwise be feasible. Interpreters generally do not perform the analysis required for most optimizations. Even within the JIT compiler, though, a full analysis can be counterproductive, as the overhead incurred does not result in performant enough code to compensate. This idea is related to the idea of client and server compilers, where there are different levels of optimization that are employed based on the expected workload. This is present in the HotSpot Java virtual machine (JVM), which has both a client (C1) and server (C2) compiler [19].

HotSpot JVM also gives a good example of the kinds of optimizations that are used in a JIT. The client compiler, for example, uses method inlining on methods which are below a certain size threshold, reduced for each nested method call [19]. Other optimizations include null check elimination, conditional expression elimination, and global value numbering [3]. Throughout their design document, they highlight how certain optimizations were chosen due to the ease at which they could be done with some of the changes to the implementation.

Another important concept that is unique to runtime compilation is deoptimizing. This occurs in HotSpot when some of the assumptions that were made during the optimization are invalidated [19]. Deoptimization is the process where these cases are detected, the optimized machine code is unloaded from memory, and control is transferred back to the interpreter. This presents an additional factor to be balanced around, as deoptimization incurs a substantial performance cost.
2.2.2 JIT Register Allocation

Register allocation is one of the major gains of using a JIT compiler over an interpreter, cutting down on costly memory access. With an interpreter, accessing any variable is likely to invoke some kind of memory access, which can result in a dramatic slowdown. In compiled settings, the use of variables can be analyzed in order to keep some commonly used values in registers on the CPU instead. For the purposes of this framework, register allocation is done by the LLVM backend, but is still conceptually important to students.

Common techniques used for register allocation in general include graph coloring algorithms, such as Chaitin-Briggs, and linear scan algorithms [16]. For JIT compilers, linear scan algorithms are typically favored over graph coloring algorithms, as graph coloring algorithms have computational complexity that is undesirable for runtime compilation [16]. Linear scan, for example, is featured in HotSpot JVM [19]. However, linear scan is a greedy algorithm, and the register allocations that it produces are typically of lower quality than those generated by graph coloring algorithms.

Register allocation, like many other optimizations, runs upon the need for a balance between up-front compilation costs and long term gains from the optimizations performed. Many JIT compilers opt for introducing "tiers" of optimizations, gradually increasing the aggressiveness of the optimizations performed as code is run more often. One example of this is with the HotSpot JVM for Java, which has two main compilers: a "client" compiler, which is intended for shorter running programs, and a "server" compiler which is intended for longer running programs like those that would be running on a server [3]. With HotSpot, the client compiler performs a linear scan register allocation, while the server compiler uses a graph coloring algorithm.
Previous works have investigated the possibility of optimizing register allocation techniques for dynamic compilation as is common with JIT compilers. Some utilize information that can only be gathered at runtime in order to guide register allocation. Eisler et al. [11] investigated performing register allocation on traces that were constructed at runtime. In other cases, the algorithms are altered in a way that could be done ahead of time, but is better suited to runtime compilation. Cooper and Dasgupta [14] created a method based upon Chaitin-Briggs graph coloring, but with a different process taken when spills occur. Their implementation outperformed linear scan on 9 out of the 11 selected benchmarks, indicating that there could be room for more sophisticated register allocation algorithms in JIT compilers.

### 2.2.3 JIT Security

Work in the field of JIT compiler security has largely been focused on memory safety. JITs present a significant risk for arbitrary code execution attacks, since they must generate native code that is executable. Attackers can utilize this to write their own code to memory regions that the JIT is interacting with, which can then be made executable and run to gain control of the program flow [5]. Other attacks, such as gadget chaining [20], can reinterpret existing and correct code in unintended ways. These attacks are commonly realized through JIT spraying, a form of heap spraying intended to bypass defense techniques like address space layout randomization (ASLR). To address this, frameworks used protections such as control-flow integrity [4] and sandboxing [7], each with different mitigations and overhead. Research has also been done into hardening compilers more generally against time attacks [8] [9]. Some of this work has extended towards JIT compilers, such as with Van Cleemput et al.’s work [9] with profile-based timing attack protections.
Table 2.1: Unpatched, secret-sensitive worker function across 1,000,000 total invocations. Time ratios closer to 1.00 indicate less susceptibility to timing attacks.

<table>
<thead>
<tr>
<th>T:F</th>
<th>T Time (s)</th>
<th>F Time (s)</th>
<th>Time ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>100:1</td>
<td>1.35</td>
<td>1.99</td>
<td>0.678</td>
</tr>
<tr>
<td>50:1</td>
<td>1.34</td>
<td>1.65</td>
<td>0.810</td>
</tr>
<tr>
<td>10:1</td>
<td>1.35</td>
<td>1.45</td>
<td>0.922</td>
</tr>
<tr>
<td>3:1</td>
<td>1.35</td>
<td>1.35</td>
<td>1.00</td>
</tr>
<tr>
<td>1:1</td>
<td>1.34</td>
<td>1.34</td>
<td>1.00</td>
</tr>
</tbody>
</table>

2.2.3.1 Experiments with Timing Variation

Another form of attack against JITs that was personally investigated by the author is in the form of a timing attack. JITs by nature change the timing of the program, as they are intended to be a speed boon. The question posed was whether the selective compilation of a JIT compiler could produce time differences in the code’s runtime that could be observed externally to determine what the path of code execution had been. The experiment conducted was with a function that was sensitive to its argument in different ways, either by an arithmetic expression or by explicit branching. This was then run through an interpreted-only environment and a JIT-compiled environment. The chosen language was Python, with the interpreted version being the base implementation, and the JIT-compiled version being the tracing JIT implementation from PyPy[18].

The abridged results are shown in Tables 2.1 and 2.2, which show the timings observed with each state of the argument.

The first column shows the ratio between the number of times the function is called with an argument of True (T) to the number of times it is called with an argument of False (F). The second and third columns list the average time for the worker function to complete in seconds for True and False arguments, respectively. The fourth column
Table 2.2: Patched worker function across 1,000,000 total invocations. Time ratios closer to 1.00 indicate less susceptibility to timing attacks.

<table>
<thead>
<tr>
<th>T:F</th>
<th>T Time (s)</th>
<th>F Time (s)</th>
<th>Time ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>100:1</td>
<td>2.51</td>
<td>3.14</td>
<td>0.801</td>
</tr>
<tr>
<td>50:1</td>
<td>2.53</td>
<td>2.84</td>
<td>0.891</td>
</tr>
<tr>
<td>10:1</td>
<td>2.53</td>
<td>2.60</td>
<td>0.974</td>
</tr>
<tr>
<td>3:1</td>
<td>2.54</td>
<td>2.54</td>
<td>1.00</td>
</tr>
<tr>
<td>1:1</td>
<td>2.55</td>
<td>2.54</td>
<td>1.00</td>
</tr>
</tbody>
</table>

lists the ratio between the two average times. A ratio of exactly 1 would indicate the function completes in the same time on average for both possible arguments, in which case the function’s running time is independent of the argument. The results from this experiment indicated that, depending on the environment, a JIT-induced timing attack could be feasible.

2.3 Overview

JITed’s relation to the related works is primarily through implementing the ideas contained in the past work. The body of research that this document builds upon guided the topics that were deemed important to teach. To the best of our knowledge, there is not a properly analogous system to JITed currently in published work.
This chapter goes over the overall structure of JITed, viewing its features at a high level. Figure 3.1 shows the main components of the framework and how they interact. The parser/lexer, interpreter, and module compiler are all standalone components that do not require student code in order to work. The JIT system as a whole incorporates two student-written components, which are required for the JIT to function.

Figure 3.1: A diagram of the system’s components. Student-written components are bolded and have rounded corners.

We will first discuss the target language that JITed is written for, Mini. After this, we will discuss the different components of JITed and their role in the system.
3.1 Mini

The following lists a specification for the syntax of the Mini language which JITed targets. The semantic specifications can be found in the appendices. The semantics of the language are slightly altered from the original specification. This is because the original details allowed for a reasonable amount of flexibility in how the language worked in order to make it more forgiving for students to develop for it. With the introduction of an interpreter, however, these details need to be fixed in order for smooth functionality to be ensured.

The syntax details are as follows. Non-terminals are denoted in **bold**, and terminals are in *typewriter* font.

\[
\text{program} \rightarrow \text{types declarations functions} \\
\text{types} \rightarrow \{\text{typeDeclaration}\}^* \\
\text{typeDeclaration} \rightarrow \text{struct id \{nested\ decl\}}; \\
\text{decl} \rightarrow \text{type id} \\
\text{type} \rightarrow \text{int | bool | struct id} \\
\text{declarations} \rightarrow \{\text{declaration}\}^* \\
\text{idList} \rightarrow \text{id \{, id\}^*} \\
\text{functions} \rightarrow \{\text{function}\}^* \\
\text{function} \rightarrow \text{fun id parameters returnType \{declarations statementList\}} \\
\text{parameters} \rightarrow (\{\text{decl \{, decl\}^*}\})_{\text{opt}} \\
\text{returnType} \rightarrow \text{type | void}
\]
statement → block | assignment | print | conditional
   | loop | delete | ret | invocation
block → {statement_list}
statement_list → {statement}*
assignment → lvalue = {expression | read};
print → print expression {endl}opt;
conditional → if (expression) block {else block}opt
loop → while (expression) block
delete → delete expression;
ret → return {expression}opt;
invocation → id arguments;
lvalue → id {. id}*
expression → boolterm {|| boolterm}*
boolterm → eqterm {&& eqterm}*
eqterm → relterm {{== | !=} relterm}*
relterm → simple {{< | > | <= | >=} simple}*
simple → term {{+ | -} term}*
term → unary {{* | /} unary}*
unary → {! | -}*selector
selector → factor {. id}*
factor → (expression) | id argumentsopt | number
   | true | false | new id | null
From the syntax, it can be derived that the supported operations are:

- Creation and assignment of variables
- The allocation and deallocation of heap memory (new, delete) for structs
- The binary operations of equality (==), inequalities (!=, <, >, <=, >=), integer arithmetic operations (+, -, *, /), and boolean operations (&&, ||)
- The unary operations of boolean negation (!) and arithmetic negation (-)
- The definition of new struct types
- Accessing fields in struct objects

3.2 System

The following sections present some of the details and arrangement of the framework’s components. The general workflow is that Mini source files are lexed and parsed into an ANTLR parse tree. This parse tree is then further processed to produce an abstract syntax tree. The abstract syntax tree is then provided as the input to both the interpreter and student-written compilers. The full JIT extends upon the interpreter, and accepts as parameters a compiler and a function to choose what code to compile by. This results in the program runtime, with the results evaluated by the complete JIT.

3.2.1 Lexer, Parser, and AST

The system begins with processing the source files into an abstract syntax tree (AST). The lexing and parsing are done via files generated by ANTLR[1], targeting C++ as
ANTLR takes a file in the form of a grammar specification and outputs a set of source files for both a lexer and parser that recognize that language. This process ensures a robust front end for the system. ANTLR provides the option of either a listener or a visitor pattern for the produced code. The implementation for JITed uses a visitor pattern.

Of note is that the visitors produced by ANTLR for C++ use a class named `Any` for their return type. `Any` is a type-erasing construct, making it difficult to preserve and use the inheritance hierarchy of the classes. A function that proxies a class as its base type to allow for uniform extraction from `Any` is included with the library, and is both used by the interpreter and accessible to the students for the same purpose of creating visitors.

Once the source has been processed through the ANTLR components, it must be processed into a more usable form. This is where the program gets processed into an abstract syntax tree. The AST is the form in which the students will access the code, and also the form that the interpreter runs off of. It represents the program in terms of a tree of statements, operations, expressions, and the like. To do this, a set of classes representing the various nodes in the AST were created, along with visitors to translate from the ANTLR context classes to the concrete AST classes.

### 3.2.2 Interpreter

The interpreter is the backbone of the framework, as it is the initial entry point for programs, and also has a hand in the decision making process for running JIT’d code. The interpreter takes the AST and processes it branch by branch, resolving the nodes into concrete values to execute the program. The interpreter is a complete
implementation of Mini, and so the framework is provided to students in a form that will be immediately able to execute programs.

In the framework, decisions on whether to run code via the interpreter or potentially running JIT’d code happen at the level of function invocations. JIT compilation in JITed is done per-function, making it so that the handoff from interpreter to JIT only occurs when new functions are called. The JIT’d code is able to call both other JIT’d functions, as well.

Of note is that the interpreter is a concrete implementation of Mini, the behavior of which should be matched by the compiler in order to prevent inconsistencies at runtime. This restricts the semantics of Mini to be more defined and thus less open for student adjustments to behavior, but is an unavoidable part of running a JIT.

3.2.3 JIT Environment

The final and most conceptually important stage of JITed is the back end JIT compiler and its corresponding API. While it may seem counterintuitive that a framework for developing JIT compilers would feature a complete one as part of its components, the purpose of this compiler is to take the student-compiled intermediate representation (IR) and make it runnable. The student JIT and the environment JIT thus act in tandem as one overall JIT compiler, with the separation solely to allow the student to freely develop their own solutions.

The compiler for this stage uses the C++ LLVM and Clang libraries, both libraries from LLVM Developer Group. Code provided from the students in string format is parsed into modules, which are the LLVM libraries’ internal way of representing sections of code. These modules are then compiled and linked into the runtime.
environment alongside some C++ library and utility functions. The utility functions provide the printing functionality and heap management functions.

JIT compilation in JITed is done on a per-function basis. This allows branching between interpreted code and compiled code to be done in a straightforward fashion, and is a useful logical division for students to work with. The logical division of code into functions allows the students to create their compilers without having to worry about the process of compiling the entire program, involving the processing of function headers, type definitions, and globals. Students are able to access the processed results of these freely within the environment. An important note is that compiling a function will also invoke compilation for all the functions that it depends upon. This was done due to limitations in the environment, but may be opaque to students if not mentioned.

The two classes of note that the JIT compiler makes use of from the framework are the module compiler and the dependency finder. The former contains the LLVM library calls necessary to make the program runnable, and also manages the symbols that the JIT will call into. The dependency finder gets the names of all functions that are called within the body of a given function in a recursive fashion. When a function is requested to be JIT-compiled, the dependency finder gets all of the targets for compilation, then once the compilation from the students is done, the module compiler turns the code into a runnable module.

### 3.2.3.1 JIT Components

The last two components are the portion of the JIT that should be written by the students: a compiler that takes Mini source and produces LLVM IR code, and a function to decide when functions should be JIT-compiled.
For the student to interact with the system, they extend upon exposed classes and provide a level of expected functionality. The two high-level constructs that students implement and provide to the JIT framework are a compiler and a “heat” function. The shell for the compiler is the ASTVisitor class, which builds upon ANTLR’s visitor pattern structures. The interface is provided in more detail in the Implementation Details. At a high level, it exposes functions to “visit” each of the objects making up the AST, and perform some function upon them. In the case of student-written code, this is a compiler that takes the AST objects and produces LLVM IR code in the form of a string. To do this, students implement the visit function for AST Functions to return the compiled code, which the JIT can then compile and link into the runtime.

The JIT also exposes two variables pertaining to the functions being called in the program. One of these is simply the most recent function that was called, and the other is a map containing the call counts of each of the functions in the program. The JIT exposes these variables to allow students to write a function that determines when a function should be JIT-compiled versus being interpreted, described as the heat function. The heat function used by the JIT can be set to one of the student’s creation. It takes as arguments the name of the function being called, the most recent function call prior to that, and the call counts of each function. The result is a boolean value representing whether the function should be compiled.

The writing of these two components presents the bulk of the work that is performed by students. Other tasks are fairly minor, including things such as setting up a main function to invoke the appropriate operations.
Chapter 4

STUDENT WORKFLOW

This chapter will briefly go over the use of JITed from a student’s perspective. Much of this document has focused on an instructor’s point of view, as the code that students write is fairly static in purpose. The compiler, written properly, will always function according to Mini’s specification, so there is no variance there. Similarly, the heat functions should be relatively straightforward, though there is room for further statistics to be incorporated to add to the complexity.

4.1 Student Use of JITed

When students first interact with JITed, there will be no active compiler, and thus the system will act as an interpreter. This is a good time for students to experiment with the Mini language itself, getting comfortable with the syntax and its behavior.

The first step for students would be to write a base compiler that supports all of the features of Mini. While the compiler is a work-in-progress, they may encounter undefined behavior or errors in the LLVM compilation if compilation is requested on a function using features they don’t yet support. Once the compiler is done, the behavior should be consistent across all students, supporting the same language with well-defined behavior.

JIT implementations will mostly be differentiated by the heat function used and the optimizations implemented. The efficiency with which the optimizations are implemented will also be of concern to students, as the optimizations are being done at
runtime. The heat functions will be the second component that students will work on, tuning the function to provide reasonable results and performance.

The final stage will be to implement optimizations and perform any final performance tweaks. Optimizations should be lightweight in nature, with time taken for compilation directly adding to a program’s runtime. Ultimately, any number of optimizations could be implemented for this. Depending on the course, students could write a small number of very fast optimizations, or implement both fast optimizations and more in-depth ones for heavier workloads.

4.2 Example of Heat Functions

This section will give a more concrete example of the heat functions that students will write as part of their implementation. The first function listed, `defaultHeatFunction`, is the default function as implemented in the JIT class itself. The two that follow are examples of what a student might write, the first saying to compile a function once it has been run at least 5 times, while the second says to compile a function if it accounts for at least 25% of the total functions called.
bool defaultHeatFunction(string called, string lastCalled, map<string, int> callCounts)
{
    return true;
}

bool threshold(string called, string lastCalled, map<string, int> callCounts)
{
    return callCounts.at(called) >= 10;
}

bool proportion(string called, string lastCalled, map<string, int> callCounts)
{
    const int minCalls = 4;
    const double prop = 0.25;
    int total = 0;
    for (auto pair : callCounts) {
        total += pair.second;
    }
    return total >= minCalls &&
            ((double)callCounts.at(called)) / total >= prop;
}

Figure 4.1: Examples of heat functions.

The performance of these functions would vary based on the programs that are run. For example, consider the function calls listed in Table 4.1. The D, T, and P columns show whether that function would be JIT compiled with heat functions of the default, threshold, and proportion functions above, respectively. In this example, the functions are called in the order they are listed in the table.

The JIT behavior between these functions differs fairly significantly. Furthermore, even with the same basic approach, adjusting parameters could result in very different decisions. For instance, the proportion function used here had a very low requirement for the minimum total calls, leading to the first function being JIT’d even though it
Table 4.1: An example of functions being called and how many times they were, in order. The rightmost three columns list whether they would be JIT’d using the default, threshold, and proportion heat functions, respectively.

<table>
<thead>
<tr>
<th>Function</th>
<th>Program 1</th>
<th>D</th>
<th>T</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun firstToBeCalled()</td>
<td>5 times</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>fun calledALot()</td>
<td>50 times</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>fun calledALittle()</td>
<td>2 times</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fun calledMore()</td>
<td>15 times</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>fun foo()</td>
<td>18 times</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

accounts for relatively few of the overall calls. A higher requirement could prevent this behavior, which may be desirable depending on the program.
4.2.1 Example of AST and Interpretation

Figure 4.2: A short example Mini program. The Mini code is in a box in the top left, while the tree on the right is the parsed AST. Edge labels indicate the field name, node labels indicate the object type.

This section presents a brief example of how the interpreter operates. Figure 4.2 contains a very short Mini program alongside its AST parse. This example does not demonstrate all of the features of Mini, but covers a reasonable amount of them without being unmanageably large.

To begin with, constructing an instance of the interpreter will iterate through the global, type, and function declarations, populating the interpreter’s maps from name
to declaration. Following this, running the interpreter runs the main function, creating its scope and declaring the local variable “foo”.

The first statement to be evaluated is an assignment statement, with the Lvalue being a simple identifier and the right side being a “new” expression. Evaluating the NewExpression results in heap memory being allocated for a PackedStruct*, and a TypedValue being created around that memory. The left-hand side gets evaluated to a pointer to the local that was created when entering main, which gets the memory address from the new expression.

The second statement is another assignment statement, this time accessing a field in the newly allocated struct. The LvalueDot evaluates to a pointer to one of the PackedStruct’s accessors, which gets the new integer value of 5 from the right-hand side.

The third statement returns the value that was set in the previous assignment statement. Note that for this statement, a DotExpression is used, as opposed to an LvalueDot. The result is a non-pointer TypedValue, which is returned from the main function. The raw return value is then extracted by MiniInterpreter::run, and returned.

Note that the heap memory allocated for the variable foo is not deleted in this program, but should be in typical code.

4.3 Compiler Interface

This section will cover the expected behavior of any compiler that the student implements, and details on why that behavior should be that way.
```objc
define void @printList(%struct.LL* %head)
{
    LU0:
    %u3 = icmp ne %struct.LL* %head, null
    br i1 %u3, label %LU2, label %LU3
    LU2:
    %phi0 = phi %struct.LL* [%head, %LU0], [%u9, %LU2]
    %u5 = getelementptr %struct.LL, %struct.LL* %phi0, i1 0, i32 0
    %u6 = load i32, i32* %u5
    call void @printIntEndl(i32 %u6)
    %u8 = getelementptr %struct.LL, %struct.LL* %phi0, i1 0, i32 1
    %u9 = load %struct.LL*, %struct.LL** %u8
    %u10 = icmp ne %struct.LL* %u9, null
    br i1 %u10, label %LU2, label %LU3
    LU3:
    br label %LU1
    LU1:
    ret void
}
```

**Figure 4.3:** An example of the kind of output that the JIT expects to be returned by a compiler.

The compiler that students write should be contained within a class that extends the ASTVisitor. Beyond this, the extensions that the student makes are up to them. The way that the JIT interfaces with the compiler is to initialize it using its constructor, and then call its visit function on the Mini function being compiled.

The return type from the visit function should be a `string`, containing the function definition only. An example of this is shown in Figure 4.3. The only function that is accessed from outside of the compiler is the `Any ASTVisitor::visit(Function*)`. This function is the only one that is required to be defined by the compiler.

The first main characteristic of note is that the output from the compiler does not contain any additional information that would ordinarily be required to fully compile to native code. This is shown in Figure 4.3, where a struct type named `%struct.LL` is used without being defined in the code snippet.
string fnString = funcs.at(fname)->accept(compiler.get());

Figure 4.4: The line in which the compiler is invoked from the JIT.

The other characteristic is that values are passed by value, i.e. the LLVM code should take as parameters the values themselves in the case of literals, and pointers to struct types. How the function handles these values beyond that is up to student discretion. Additionally, the boolean types are expected to be 32-bit ints (i32), as are integer values. Values should be zero-extended and truncated as needed to fit this.

Finally, in order to access input/output functions, three predefined functions are linked and should be used by the student code. These are `printInt`, `printIntEndl`, and `readInt`. The first two print integers to stdout, while the last reads an integer from stdin.

The ASTVisitor class provides an empty shell to the student to write their compiler. Should this not be desired, modifications could be made fairly easily in order to use a more abstract class.
Chapter 5

IMPLEMENTATION DETAILS

This chapter discusses the details of the implementation. The full code is available online at https://github.com/ccwatts/JITed. Namespace qualifiers are omitted in the descriptions below, where unambiguous. This means that the names of things may have prefixes jited::, jited::ast::, minic::, std::, or others that are not shown in this document, for the sake of readability.

Throughout this section, classes will appear in the form of ClassPtr, where Class is one of the classes that have been defined. These are typedefs of std::shared_ptr<Class> used for readability.

5.1 Abstract Syntax Tree Details

The primary construct that the system interacts with is the abstract syntax tree (AST). The AST represents the program as a set of operations and their operands, with non-terminal nodes to represent the operations and edges between them determining the meaning. Terminal nodes can then represent the variables and values in the program. The AST provides a useful form for the interpreter to work with, and also the input for the compiler to construct a control flow graph (CFG) with.

Within the code, the AST is represented by a set of classes. At the highest level, the first node is always a Program node, which contains information about the functions, structs, and globals available. The nodes comprising the program itself are broken into types, expressions, and statements, which represent the types values can have,
the values and operations that can be done on them, and the statements operating on those values, respectively.

It is important to note that the language that the AST is describing is Mini, and not LLVM IR at this point. As such, the types used and the supported operations are C-like and generally more abstract than in LLVM. The classes described here are contained within the `jited::ast` namespace.

5.1.1 Types

Type objects are used to give details about the type that values hold in the program. Objects are used to allow additional functionality to be associated with types, and to allow struct types to be described in detail. In terms of the implementation of further systems, types are an important construct but relatively low impact in terms of how they are processed. Generally, the only information required of a type is what its name is, which allows for comparing the types of values and printing types in LLVM IR.

JITed contains the following Type objects:

- **Type**, the base class.
- **BoolType**, for boolean values.
- **IntType**, for numerical values.
- **VoidType**, used only for function return types.
- **StructType**, for struct types, which includes the name of the struct within it.
5.1.2 Expressions

The Expression classes all correspond to the Mini expressions, which cover the basic primitives, structs, and various operations that combine or modify them. Many of the statements contained within Mini are simply wrapper around an expression which carries the underlying operation. Expressions are often nested, with objects like BinaryExpressions containing their operands, which are also Expressions themselves.

JITed contains the following Expression objects:

- **Expression**, the base class for all right-hand side expressions.
- **BinaryExpression**, for all of the arithmetic, boolean, and comparison operations.
- **DotExpression**, for struct access only. Struct assignment uses **LvalueDot**, which mirrors the functionality.
- **TrueExpression** and **FalseExpression**, used for the literal boolean values of true and false.
- **IdentifierExpression**, used for variables identified by name.
- **IntegerExpression**, used for literal integer values.
- **InvocationExpressions**, used for function calls.
- **NewExpression**, used to allocate a new struct in heap memory.
- **NullExpression**, used for literal null values.
- **ReadExpression**, used for reads from standard input. The result of a read is strictly an integer.
• **UnaryExpression**, used for arithmetic negation (-) and boolean negation (!).

In addition to expressions, there are separate classes distinguished as Lvalues, which are strictly used for the left-hand side of assignments. Two concrete classes exist from this: **LvalueId**, for assigning to a variable identified solely by the variable name, and **LvalueDot**, used when assigning to a member of a struct.

### 5.1.3 Statements

The Statement classes correspond to the different statements in Mini. Similar to the Expressions, they do not contain much functionality themselves; they primarily exist as a vehicle for the visitor pattern. Three other classes are included in this section, as the information in them is typically contained in a single line. These are the Declaration, TypeDeclaration, and Function classes.

• **Declaration**, used for the declaration of variables, i.e. globals, locals, and both parameters and arguments. These do not include definitions.

• **Function**, used to describe a function’s details: the line of its header, its name, return type, parameters, locals, and body.

• **TypeDeclaration**, used for declarations of new struct types, containing the name of the struct and all of the fields’ names and types.

• **Statement**, the base class for the following statement classes.

• **AssignmentStatement**, used for assigning variables and struct members.

• **BlockStatement**, used for brace-enclosed blocks of code. In effect, this is an ordered list of instructions.
• **ConditionalStatement**, used for conditional ("if") statements.

• **DeleteStatement**, used for the deallocation of structs contained in heap memory.

• **InvocationStatement**, used for invocations of functions that are not being assigned to variables; that is, a function call by itself on a line.

• **PrintStatement** and **PrintLnStatement**, used for printing variables to stdout. PrintStatement includes a space after the variable; PrintLnStatement includes a newline (\n).

• **ReturnStatement** and **ReturnEmptyStatement**, used for returning a value (ReturnStatement) or nothing (ReturnEmptyStatement) from functions. A function with a void return type must only use ReturnEmptyStatements, while one with a non-void return type must use ReturnStatements.

• **WhileStatement**, used for while loops, which are Mini’s only form of loop.

### 5.1.4 Other Constructs

The other AST classes within the framework are **Program**, which contains vectors of all of the struct types declared, the functions, and global declarations. Aside from this, there are three visitor classes: one for all of the AST classes (**ASTVisitor**), one for just the expressions (**ExpressionVisitor**), and one for statements and classes like Program (**StatementVisitor**). These visitor classes are not abstract, but the default implementation of each of them simply throws a runtime error exception.
5.2 Interpreter Details

This section goes over the details of the interpreter in JITed. Since the JIT class extends upon the interpreter, this section also has major implications in terms of how the JIT framework itself functions. The interpreter is a concrete implementation of the Mini specification; this means that the specification that students must implement must match this concrete implementation. Instances where restrictions are imposed beyond the initial specification of Mini will be noted in the sections that follow.

The first subsection will discuss some of the common data structures that are manipulated by the interpreter and a required workaround for working with ANTLR’s Any, used by ANTLR to emulate virtual templated functions. The second subsection will discuss the interpreter proper.

5.2.1 Typed Values

The primary data structure used in the compiler to represent values is the TypedValue class, in the jited:: namespace. This class is a container for another value of arbitrary type; for any struct values, another class called PackedStruct is used. To understand the structure of the TypedValue class, we must first briefly discuss ANTLR’s Any class.

5.2.1.1 antlrcpp::Any

ANTLR’s Any class is used to circumvent C++’s restriction that templated function not be virtual, as they are in the Java ANTLR interface. Any is, as its name suggests, a container class for any value, i.e. a type erasing construct. Anys themselves do not store any information on the type of data contained. Checks for the type of data
contained must be done using the templated `bool Any::is<T>()` function. This function attempts a cast using C++’s `dynamic_cast` and checks for an exception in order to determine whether the contained data is a valid instance of the template class. The other important function it has is the `T Any::as<T>()` function, which is used to extract the contained data given that you know its type. This is akin to casting a void pointer to a pointer of a known type.

One important aspect of Any is that it erases any information about polymorphism. This means that attempting to implicitly upcast by extracting a value as its base class results in an exception. For example, in the code provided in Figure 5.1 which mirrors aspects of the interpreter’s implementation, performing an otherwise valid upcast results in an exception.

```cpp
class Foo {};  
typedef std::shared_ptr<Foo> FooPtr;

class Bar : public Foo{};  
typedef std::shared_ptr<Bar> BarPtr;

void fun() {
    BarPtr b = std::make_shared<Bar>();
    antlrcpp::Any a = antlrcpp::Any(b);
    FooPtr f = a.as<FooPtr>();// exception is thrown
}
```

**Figure 5.1:** Potentially counterintuitive behavior: the line where `f` is defined throws an exception, whereas if there were not the intermediate usage of Any, it would work.

To work around this, JITed includes a `proxy` function that wraps a given class in a shared pointer of its base class, then places that value in the Any. The header for this function is `antlrcpp::Any proxy(std::shared_ptr<U> from)`, with template variables of `U`, the derived class, and `V`, the base class. A working version of the
code from Figure 5.1 that utilizes the proxy function is shown in Figure 5.2. The contents of the function are included in a comment for reference.

```cpp
#include "lib/proxy.h"

/*
template <typename U, typename V>
antlrcpp::Any proxy(std::shared_ptr<U> from) {
    auto proxied = std::static_pointer_cast<V>(from);
    return antlrcpp::Any(proxied);
}
*/

class Foo {};
typedef std::shared_ptr<Foo> FooPtr;

class Bar : public Foo{};
typedef std::shared_ptr<Bar> BarPtr;

void fun() {
    BarPtr b = std::make_shared<Bar>();
    antlrcpp::Any a = jited::proxy<Bar, Foo>(b);
    FooPtr f = a.as<FooPtr>(); // works
}
```

Figure 5.2: Functional upcasting of values contained within an antlrcpp::Any. The Any itself should be constructed using the proxy function. Extraction can then be done normally.

Now that the framework’s use of Any has been detailed, we can discuss the main class used for evaluated values: TypedValue. A description of it is provided in Table 5.1.

Table 5.1: Description of the TypedValue class.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
</table>

37
<table>
<thead>
<tr>
<th>Any value</th>
<th>The raw value contained within the object. This may be any variety of types, including points.</th>
</tr>
</thead>
<tbody>
<tr>
<td>string type</td>
<td>A string containing the name of the Mini type that the object holds. This will almost always differ from the name of the C++ type contained in the value.</td>
</tr>
<tr>
<td>bool initialized</td>
<td>A boolean indicating whether the value has been initialized; when the variables are first declared, this will be false. Only the setValue function sets this to true; it is sometimes necessary for value to be set manually, but in practice this should be done very sparing.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;typename T&gt; T as()</td>
<td>Returns value as the given type. This function is a wrapper around Any’s own as&lt;T&gt;() function.</td>
</tr>
<tr>
<td>bool isStruct()</td>
<td>Returns whether the contained value is a struct.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>void setValue(antlrcpp::Any val, PackedStruct* parentStruct=NULL, std::string lastField=&quot;&quot;)</code></td>
<td>Sets <code>value</code> to the provided value. If the object’s <code>value</code> is a pointer type, this will not change the pointer, but set the value at that memory location to the raw value provided. The second and third arguments are to be able to set up proper parity when setting a struct’s field.</td>
</tr>
<tr>
<td><code>int asInt()</code></td>
<td>Returns the contained value as an int. This will dereference any integer pointers, and is the preferred way to get any integer value out of a TypedValue.</td>
</tr>
<tr>
<td><code>bool asBool()</code></td>
<td>Returns the contained value as a bool. Similar to <code>asInt()</code>, this dereferences any pointers in the contained value.</td>
</tr>
<tr>
<td><code>int32_t asI32()</code></td>
<td>Returns the contained value as an signed int that is guaranteed to be 32 bits in width. This will dereference any pointers, and will explicitly work in the case of both bool and int values.</td>
</tr>
<tr>
<td><code>PackedStruct* asStruct()</code></td>
<td>Returns the contained value as a PackedStruct pointer. This is only valid when the contained value is actually a pointer to a PackedStruct.</td>
</tr>
<tr>
<td><code>bool isNull()</code></td>
<td>Returns whether the contained value is both a struct value and is null.</td>
</tr>
</tbody>
</table>
string toString()  
Returns a printable string representation of the contained value. This is according to Mini’s type system.

TypedValue is used for two reasons: to tie together values and types so they can be returned together, and to serve as one consistent return type for any function that returns a value in the interpreter. A visit function for an expression type can be expected to return a TypedValue. For statements, the return type is generally a `nullptr`, unless it is a return statement. Return statements do return a TypedValue, both Return and ReturnEmpty. This is used to both detect when a function should stop execution and return, and to pass the return value through the system.

JITed’s use of structs requires some changes both in terms of the compiler’s representation and the interpreter’s representation of the structs in memory. The compiler side will be discussed later along with the JIT features, but the most significant change is that all structs are packed. This means that the structs contain no padding; each value in the struct begins exactly when the other ends, as if it were an array. This is done to ensure that the system is portable, regardless of the compiler toolchain used to build JITed, and that data can be safely shared among native and interpreted code.

The interpreter uses a wrapper around a byte buffer (`uint8_t*`), named PackedStruct. A description of the PackedStruct class is provided in Table 5.2.

Table 5.2: Description of the PackedStruct class.

<table>
<thead>
<tr>
<th>PackedStruct</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td></td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
</tr>
<tr>
<td>map&lt;string, int&gt; offsets</td>
<td>A map from field names to offsets within the underlying struct.</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>map&lt;string, string&gt; types</td>
<td>A map from field names to type names.</td>
</tr>
<tr>
<td>map&lt;string, TypedValuePtr&gt; accessors</td>
<td>A map from field names to TypedValues which can be used by the interpreter in other contexts. The TypedValues themselves contain pointers to the appropriate data type, hence the importance of the TypedValue::setValue function.</td>
</tr>
</tbody>
</table>

### Function

| template <typename T> T get(string fieldName) | Returns the requested field specified by fieldName as the given C++ type. |

### Public

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static map&lt;uint8_t*, PackedStruct*&gt; lookupTable</td>
<td>A static map from raw buffers to the PackedStruct that contains it. This is used to determine when constructing a PackedStruct is necessary when using data that was produced by native code. If an entry does not exist in the table, then it was created by native code and must be wrapped for use by the interpreter.</td>
</tr>
<tr>
<td>size_t totalBytes</td>
<td>The number of bytes that the buffer inhabits.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>bool has(string fieldName)</td>
<td>Returns whether the struct in question has a field named fieldName.</td>
</tr>
<tr>
<td>TypedValuePtr at(string fieldName)</td>
<td>Returns a pointer to the TypedValue accessor that provides access to the field given by name.</td>
</tr>
<tr>
<td>int32_t getInt(string fieldName)</td>
<td>Return a raw int32_t contained in the field given by name.</td>
</tr>
<tr>
<td>uint8_t* getPtr(string fieldName)</td>
<td>Returns a byte pointer contained in the field given by name.</td>
</tr>
<tr>
<td>void setMember(string fieldName, PackedStruct* member)</td>
<td>Sets a member struct field. This only needs to be used in the case when setting a struct field inside of a struct, as there is external linkage that needs to be set up to ensure parity between interpreted and native code.</td>
</tr>
</tbody>
</table>

PackedStructs essentially contain two interfaces by which they are used. The first is the raw buffer: this is passed to native code. Any struct members in this buffer are pointers to raw buffers. This is necessary for the native code to work; it expects that a pointer to a struct will be a pointer to a packed struct type, not a C++ object. Generally, this is done simply by retrieving the buffer itself. The second interface is to use the at function, which is done in the interpreter. This returns wrappers that effectively provide a view into the packed struct underneath. Each of the TypedValues
used in a PackedStruct have pointer data types inside. These are all pointers into the buffer.

```haskell
struct Foo {
    int x;
    int y;
    struct Foo* next;
}
fun main() int {
    struct Foo a;
    struct Foo b;
    a = new Foo;
    b = new Foo;
    a.next = b;
}
```

Figure 5.3: An example of how PackedStructs are linked internally. Code that generates this example is shown in the top left.

Figure 5.3 shows how structs are nested within each other. In the case of nested structs, links to the next struct are different between the PackedStruct’s representation and the buffer’s representation. The accessor in the PackedStruct points to the next PackedStruct; the pointer within the buffer points to that PackedStruct’s `buffer`.

This concludes the data structures that are used by the interpreter internally. TypedValues are the primary object being manipulated, with PackedStructs sometimes being the contents of those TypedValues.

### 5.2.2 Interpreter

The majority of the functions in the interpreter class proper are simply visit functions, which are invoked when `accept` is called from an AST object. The results of these
functions are, to an extent, self-explanatory. The focus in the description of these functions will be on any important notes on side effects or branching behavior. The description of the MiniInterpreter class is given in Table 5.3.

**ValueMap** below is a typedef of `map<string, TypedValuePtr>`.

Table 5.3: Description of the MiniInterpreter class.

<table>
<thead>
<tr>
<th>MiniInterpreter : ASTVisitor</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Protected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
</tr>
<tr>
<td>Description</td>
</tr>
</tbody>
</table>

| ValueMap globals |
| A map of the global values of the function. This is separate from the nested scopes used for functions. |

| PackedStruct* parentStruct |
| A variable used to keep track of the last struct that was referenced. This is used for setting struct fields inside structs. |

| string lastField |
| Related to parentStruct, this keeps track of the last field name that was used. |

| vector<ValueMap> scopes |
| A vector of scopes (mappings from string names to values). The end of the vector is the topmost scope. |

| map<string, FunctionPtr> funcs |
| A map from function names to function objects. This is used to retrieve metadata about functions. |
| **map<string, TypeDeclarationPtr> structs** | A map from struct names to the type definition. The type definitions are used when instantiating a struct of the given type. |
| **ProgramPtr program** | A pointer to the program that is being evaluated. |

<table>
<thead>
<tr>
<th><strong>Function</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bool isBool(string eType)</strong></td>
<td>Returns whether the string type name is the same as either TrueExpression or FalseExpression, using run-time type information.</td>
</tr>
<tr>
<td><strong>bool getBoolState(TypedValue&amp; tv)</strong></td>
<td>Gets a boolean value from the given TypedValue, regardless of underlying data type. More permissive with the typed value's type than TypedValue::asBool.</td>
</tr>
<tr>
<td><strong>TypedValuePtr lookup(string name)</strong></td>
<td>Looks up a typed value pointer in the current scopes and globals.</td>
</tr>
<tr>
<td><strong>ValueMap bindArgs(vector&lt;ExpressionPtr&gt;&amp; args, vector&lt;DeclarationPtr&gt; params)</strong></td>
<td>Creates a new scope for a function and binds the arguments to the appropriate parameter names.</td>
</tr>
<tr>
<td><strong>void resetStruct()</strong></td>
<td>Resets the struct-related member variables that the MiniInterpreter has.</td>
</tr>
</tbody>
</table>

**Public**

<p>| <strong>Function</strong> | <strong>Description</strong> |</p>
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int run()</td>
<td>Runs the program that the interpreter was initialized with, and returns its exit code in int form.</td>
</tr>
<tr>
<td>Any visit(AssignmentStatement* statement)</td>
<td>Evaluates an assignment statement, setting the value in the target TypedValue to the result of the right-hand side. The target TypedValue is not changed; only the contained value is. Invokes TypedValue::setValue. Returns nullptr in all cases.</td>
</tr>
<tr>
<td>Any visit(BinaryExpression* expression)</td>
<td>Returns the result of the binary expression when evaluated. Returns a TypedValue when successful; when unsuccessful, throws an exception or returns nullptr.</td>
</tr>
<tr>
<td>Any visit(BlockStatement* statement)</td>
<td>Evaluates each of the statements within the given block statement. If any statements return a non-null value, execution stops and that non-null value is returned from this function.</td>
</tr>
<tr>
<td>Any visit(BoolType* type)</td>
<td>Returns the string name of the boolean type, i.e. BoolType::name().</td>
</tr>
<tr>
<td>Any visit(ConditionalStatement* statement)</td>
<td>Evaluates the conditional statement, invoking accept on the appropriate branch depending on the result of the guard.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>Any visit(Declaration* declaration)</td>
<td>Processes the declaration of a new variable. This allocates a new shared pointer to a TypedValue and returns it.</td>
</tr>
<tr>
<td>Any visit(DeleteStatement* statement)</td>
<td>Deallocates the struct contained in the delete statement. This will throw an exception if the value being deleted is not a struct, or under the normal circumstances where delete would fail. Returns nullptr.</td>
</tr>
<tr>
<td>Any visit(DotExpression* expression)</td>
<td>Evaluates the left-hand side of the struct access, then if it is a valid struct, attempts to retrieve the field given. Throws an exception if the left-hand value is not a struct, is null, or uninitialized, or if the struct does not contain the requested field. Returns a TypedValue.</td>
</tr>
<tr>
<td>Any visit(FalseExpression* expression)</td>
<td>Returns a TypedValue containing false.</td>
</tr>
<tr>
<td>Any visit(Function* function)</td>
<td>Evaluates each of the local variable declarations inside the function, adding them to the top scope, then calls into the function body. This does not bind the arguments, nor does it create the new scope for the function; that is done at the invocation level. Returns a TypedValue containing the returned value from the function.</td>
</tr>
<tr>
<td>Any visit(IdentifierExpression* expression)</td>
<td>Evaluates the identifier (i.e. variable) expression, returning a TypedValuePtr. This will be nullptr if it is not found, or a pointer to the TypedValue for the given variable if it is.</td>
</tr>
<tr>
<td>Any visit(IntegerExpression* expression)</td>
<td>Returns a TypedValue containing the integer value in the integer expression.</td>
</tr>
<tr>
<td>Any visit(IntType* type)</td>
<td>Returns the string name of the int type in Mini.</td>
</tr>
<tr>
<td>Any visit(InvocationExpression* expression)</td>
<td>Evaluates the arguments by value, evaluating them in the current scope then binding them using bindArgs. Returns a TypedValue containing the appropriate return value for the function.</td>
</tr>
<tr>
<td>Any visit(InvocationStatement* statement)</td>
<td>Evaluates the invocation expression contained in the statement, discarding the return value, if any. Returns nullptr.</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Any visit(LvalueDot* lvalue)</td>
<td>Returns a <strong>pointer</strong> to the field being accessed. This gets the result of the accessor from the PackedStruct. Returns a TypedValuePtr.</td>
</tr>
<tr>
<td>Any visit(LvalueId* lvalue)</td>
<td>Returns a <strong>pointer</strong> to the variable being accessed. Returns a TypedValuePtr.</td>
</tr>
<tr>
<td>Any visit(NewExpression* expression)</td>
<td>Creates a TypedValue containing a newly allocated PackedStruct on the heap. Returns a TypedValue.</td>
</tr>
<tr>
<td>Any visit(NullExpression* expression)</td>
<td>Returns a TypedValue containing <strong>NULL</strong>, casted as a PackedStruct*. The initial type for this is its own type denoted by <code>Type::nullName()</code>, and not a struct type.</td>
</tr>
<tr>
<td>Any visit(PrintLnStatement* statement)</td>
<td>Evaluates the expression being printed, then prints it to stdout with a newline character following it.</td>
</tr>
<tr>
<td>Any visit(PrintStatement* statement)</td>
<td>Evaluates the expression being printed, then prints it to stdout with a space following it.</td>
</tr>
<tr>
<td>Any visit(Program* program);</td>
<td>Resets the current scopes so only one, empty scope exists, then calls into main, if one exists. Errors are printed and a nonzero return value is given if proper main function does not exist, does not return, or does not return an int.</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Any visit(ReadExpression* expression)</td>
<td>Returns a new TypedValue containing an integer value read from stdin. If the read fails, it defaults to 0.</td>
</tr>
<tr>
<td>Any visit(ReturnEmptyStatement* statement)</td>
<td>Returns a TypedValue containing <code>nullptr</code>, with type name <code>VoidType::name()</code>. This value will not be usable, but indicates that the function should return.</td>
</tr>
<tr>
<td>Any visit(ReturnStatement* statement)</td>
<td>Evaluates the expression being returned, then returns it as a TypedValue. This value should always be usable, as opposed to ReturnEmptyStatement.</td>
</tr>
<tr>
<td>Any visit(StructType* type)</td>
<td>Returns a string containing the name of the struct type.</td>
</tr>
<tr>
<td>Any visit(TrueExpression* expression)</td>
<td>Returns a TypedValue containing <code>true</code>.</td>
</tr>
<tr>
<td>Any visit(TypeDeclaration* typeDeclaration)</td>
<td>This does not do anything except return a <code>nullptr</code>. The initialization of the struct types needs to be done outside of the visitor structure in order for the timing to work for the JIT.</td>
</tr>
</tbody>
</table>
Any visit(UnaryExpression* expression)

Evaluates the given unary operation, then returns a TypedValue containing the result. Will throw an exception if the operand has an improper type for the operation.

Any visit(VoidType* type)

Returns a string containing the name of the void type.

Any visit(WhileStatement* statement)

Evaluates the guard of the while statement, then execute the body if true, and repeat. Exits early if the body returns a TypedValue, in which case it returns the same TypedValue. Otherwise, returns nullptr.

As it needs to process all parts of a program, the MiniInterpreter class has implementations for each component of the AST. The general principle of the interpreter is that statements should return a nullptr unless the current Mini function being executed has reached a return statement. Both variants of the return statement return literal TypedValues. Any statements that themselves contain further statements, like blocks, conditionals, or while statements, will pass this information along. This pattern should be followed, even if the other behavior of the interpreter is altered.

The other side of that pattern is that all expressions, when evaluated, return a TypedValue or, in the case of Lvalues, a pointer to a TypedValue. This is why Lvalues were made distinct from other expressions: in the interpreter, setting a variable means getting a pointer to what you want to alter, and then changing it in-place. The assumption that all expressions return a TypedValue or pointer to a TypedValue when
evaluated should not be violated; in general, there are no checks for whether the result is a TypedValue, as error states generally throw exceptions rather than returning an error value.

5.3 JIT Details

This section will cover the details of the overall JIT framework’s operation. The JIT class itself inherits the vast majority of its functions from the MiniInterpreter class. The only function that is overwritten from the MiniInterpreter class is the visit function for InvocationExpressions, which is not relisted here. The distinction is that the JIT’s implementation will call into the compiled function if available, otherwise it will fall back on MiniInterpreter’s implementation. It will also check whether to compile the function in question afterwards via the heat function. A description of the JIT class is listed in Table 5.4.

Table 5.4: Description of the JIT class.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unique_ptr&lt;ModuleCompiler&gt; mc</td>
<td>A pointer to the ModuleCompiler, which handles all native code generation and running.</td>
</tr>
<tr>
<td>unique_ptr&lt;DependencyFinder&gt; df</td>
<td>A pointer to the DependencyFinder, which resolves the functions that a specific function uses, directly or indirectly.</td>
</tr>
<tr>
<td>shared_ptr&lt;ASTVisitor&gt; compiler</td>
<td>A pointer to a compiler built from the ASTVisitor class.</td>
</tr>
<tr>
<td>string lastCalled</td>
<td>The name of the last function that was called.</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>map&lt;string, int&gt; callCounts</td>
<td>A map from function name to how many times it has been called.</td>
</tr>
<tr>
<td>function&lt;bool(string, string, map&lt;string, int&gt;)&gt; heatFunction</td>
<td>The function that determines whether to compile a function after it is invoked.</td>
</tr>
</tbody>
</table>

**Public**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PackedStruct* reverseLookup(uint8_t* buf, string structName)</td>
<td>Performs the lookup to find an existing PackedStruct, or creates one if none are found. This ensures that the interpreter can use structs created in native memory.</td>
</tr>
<tr>
<td>string structsString()</td>
<td>Returns a string with declarations for all the struct types in the program. These are declared packed, using both angle brackets and curly braces to surround the member types.</td>
</tr>
<tr>
<td>string globalString()</td>
<td>Returns a string with declarations for all of the globals in the program. These are declared <code>external</code> and <code>dso_local</code>.</td>
</tr>
<tr>
<td>string declareString(string fname)</td>
<td>Returns a string with the external declarations required for the function specified by <code>fname</code>.</td>
</tr>
<tr>
<td>string moduleString(string fname)</td>
<td>Combines all of the relevant strings into one string that can be fully compiled.</td>
</tr>
</tbody>
</table>
string entryFunction(string name)  
Creates an LLVM function for the JIT to use to call into. The entry function takes a single byte buffer as arguments, then parses them and calls into the function proper.

void compileFunction(string fname)  
Invokes the compiler object to compile the function body, then assembles an LLVM string and compiles it. The result is linked into the runtime.

void makeGlobals()  
Defines new memory for the global values that can be shared between native and interpreted code.

void setHeatFunction(function<bool(string, string, map<string, int>>) newHeatFn)  
Sets the function that chooses which functions to compile to a user-defined one.

static void initialize()  
Calls all of the LLVM library initialization functions required for the JIT to function.

The majority of the functionality that is contained within the JIT class, rather than one of its components, is to add all of the necessary information to compile a piece of LLVM IR code. Included in this are the declarations of globals, other functions, and structs.

One of the most important pieces that the JIT adds to the LLVM IR string before compilation is the entry function. This entry function is only called when moving from the interpreter to native code; calls from native code to other native code will call
directly into the function being called. The purpose of the entry function is to provide a function to the interpreter that is invariant with regards to the number and types of parameters the function takes. This is because at the point where the function is called by the interpreter, it must be cast to a C++ type, making it impossible to enumerate all possible combinations. The entry functions are split into cases solely based on return type, with the sole parameter being an \texttt{int32_t*} buffer.

The entry function contains a single block that simply unpacks the buffer into individual variables, then calls into the function itself. This is analogous to the process of extracting the fields of a struct; the buffer, in essence, is an on-the-fly struct.

The two other major components, aside from the student compiler, are the Module-Compiler and DependencyFinder. The module compiler manages the runtime aspects of the JIT, while the dependency finder resolves the functions needed to execute a certain function. The dependency finder is relatively simple, as the extent of its functionality is recursing through the AST and finding what functions a certain function needs in order to execute. This is not intrinsic to the JIT’s function, but rather a necessary compromise for it to work in its current configuration. The module compiler will be discussed in more detail in the next section.

5.3.1 Native/Interpreted Synchronization

Some of the decisions in the interpreter were made because of the need for both sides of the overall runtime to access the same variables. Given Mini’s scoping rules, the scope in any given function is limited to its arguments, its locals, and the globals. Of these, arguments and locals can be used by value, with alterations only happening in the given scope. The structs and globals, however, require special handling.
For globals, the solution is relatively straightforward. The normally stack-allocated value held within the TypedValues for globals are changed to be dynamically-allocated heap memory, to which the address can be safely shared. The addresses of the contained memory itself is added into the runtime, at which point it can be used directly by name in the LLVM IR.

For structs, the system uses the raw buffer of the PackedStruct object. This is the reason why the interface for the PackedStruct class wraps around a singular buffer and maintains separate references. Since the underlying buffer is set up to contain the most basic forms of each type, nested structs and structs in general can operate normally. Structs that are returned from a compiled context will be wrapped in a PackedStruct once control returns to the interpreter.

### 5.3.2 Module Compiler

The module compiler handles all of the functionality for native compilation and symbol resolution. Initialization should always be done via the static `create` function, since the ModuleCompiler needs ownership of the pointers it uses. In this implementation, `loadCommon` loads a very small subset of the C standard library and some custom utility functions. This could easily be expanded if desired, such as to allow further C functions. A description of the ModuleCompiler class is given in Table 5.5.

<table>
<thead>
<tr>
<th>ModuleCompiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
</tr>
<tr>
<td>Variable Description</td>
</tr>
</tbody>
</table>

Table 5.5: Description of the ModuleCompiler class.
| unique_ptr<llvm::orc::LLJIT> jit | The "last-mile" compiler from the LLVM library. This takes the LLVM IR code and compiles and maintains a runtime for it. |
|llvm::orc::MangleAndInterner mangler | An object that mangles the names and adds them to the JIT runtime. |

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static unique_ptr&lt;ModuleCompiler&gt; create()</td>
<td>Initializes LLJIT and MangleAndInterner objects, then if successful, creates a ModuleCompiler and returns a pointer to it. If this fails, returns a nullptr.</td>
</tr>
<tr>
<td>void loadCommon()</td>
<td>Loads necessary library functions into the JIT runtime. Included are C’s printf, malloc, and free, and the input/output functions mentioned previously (readInt, printInt, printIntEndl).</td>
</tr>
<tr>
<td>int addFile(char* fname)</td>
<td>Reads in the contents of a file specified by fname, then attempt to compile it as LLVM IR. This adds the compiled function to the JIT runtime if successful.</td>
</tr>
<tr>
<td>int addString(string input)</td>
<td>Attempts to compile the contents of a string as LLVM IR. This adds the compiled function to the JIT runtime if successful.</td>
</tr>
</tbody>
</table>
void* getSym(string name) | Returns the address of the function named name as a void pointer. Returns NULL if the function is not found.

5.4 Summary

This completes the design and implementation of the JITed system. The system features both a plain interpreter, with full implementation of the Mini specification but no JIT, and a JIT framework that utilizes a student compiler and the LLVM library. Globals and structs are shared across the two runtimes, allowing compiled and un-compiled code segments to share data.
Chapter 6

EVALUATION

For the evaluation of the framework and its capabilities, we propose the general form of a term project that students could complete. We will link aspects of this project to the framework, and highlight where features within JITed correspond to the requirements stated at the beginning of this thesis. As a reminder, the requirements are as follows:

- **REQ1**: Students should be able to identify and designate hot (i.e. frequently used) sections of code.
- **REQ2**: Students should be able to compile from minimal LLVM intermediate representation code.
- **REQ3**: Students should be able to run compiled code when desired.
- **REQ4**: The interpreter should function opaquely to the user, without direct access.

This chapter begins with an overall outline of a project, and then continue into specific examples for each of the requirements. Example exercises are also included to give concrete examples of how the requirements are fulfilled.

6.1 Project Outline

For our evaluation, we will consider a term project over the course of 10 weeks. The beginning of the project will cover the same topics as a course strictly on compilers
would. This means the basics such as control flow graphs, LLVM syntax, and handling the data types used. The JIT portion would include using the framework, optimizing in the JIT environment, and tuning the optimizations and heat function to a specific workload (i.e., client vs server workloads). Table 6.1 lists a proposed timeline and order for the topics to be presented.

| Table 6.1: Estimated timeline of the term project. |
| --- | --- |
| **Milestone** | **Estimated Week of Completion** |
| Using the JITed Interpreter | Week 1 (1 week long) |
| Static Type Checking | Week 2 (1 week long) |
| Compiling to LLVM IR | Week 4 (2 weeks long) |
| Using a Heat Function | Week 5 (1 week long) |
| Optimizations | Week 8 (3 weeks long) |
| Tuning Optimizations for Workload | Week 10 (2 weeks long) |

With this timeline, working with JIT-specific elements accounts for roughly half of the total duration of the project. In practice, this would need to be adjusted in order to better fit the lecture material. The timeline presented simply represents a suggested project outline.

### 6.2 Fulfillment of Requirements

This section discusses each of the requirements in detail, and presents exercise to demonstrate the fulfillment of each requirement.

#### 6.2.1 Requirement 1: Identifying Hot Code

The first requirement is that students be able to identify for themselves what code should be JIT compiled, and what metric to use for this. The default function for this task simply returns true in all cases, indicating it should be compiled. The student
functions could take approaches such as a call threshold, the compilation status of
the function it was called from, or the proportion of total calls a function comprises.
This is fulfilled by the use of an explicit heat function, which can be swapped out as
desired by the student. For the base implementation presented alongside this thesis,
only two basic statistics are exposed, the function called immediately before the one
being evaluated and the call counts for each function thus far. This could easily be
expanded, with the heat function itself being adjustable by changing the type of the
heat function member variable in the JIT.

An example of an exercise touching on this concept would be providing code with a
great number of functions, called in varying amounts. What is considered a solution
or successful could be judged either by the performance of the JIT’d program, if
done late in the course, or by correctly identifying certain high priority functions in
a program.

In the case of this exercise using JIT’d code performance as a metric, it is important
that a high volume of different functions be called, in order to accumulate as much
compilation time as possible. This, alongside a small number of functions being called
far more than others, ensures a significant performance difference between a properly
formed heat function and an improper one.

### 6.2.2 Requirement 2: Compiling from Minimal LLVM

The second requirement is twofold: that LLVM IR be an acceptable input to be
compiled, and that the LLVM IR given by the student should only require the function
definition itself. This is to reduce the overall development time of the compiler, giving
more time for learning JIT concepts. This additionally allows time for there to be
individual labs and exercises interacting with a JIT, whether that be for the project itself or additional labs besides.

As an informative exercise, the full code that is actually compiled can be printed or otherwise shown to the student. The syntax used and the various keywords are still of importance, even though in the context of this language and project it is largely a rote task. If done, this should be early into the process of building the JIT compiler, to prevent students from including redundant information in their LLVM IR and expending unnecessary effort.

6.2.3 Requirement 3: Running Native Code

The third requirement is that students should be able to run compiled code freely. This is a statement of the student having control of whether to run code as interpreted, or compile it. In that case where the programs being run are known ahead of time, the heat function can be specifically set up to prevent functions from being compiled. Additionally, further extensions to Mini’s syntax could include flags on functions as hints for the JIT.

The exercises that tie in with this requirement are, effectively, the later portion of the term project. As the project goes on, more and more functions should be able to be compiled, should the student choose to do so.

6.2.4 Requirement 4: Standalone Interpreter

The fourth requirement is simply that the interpreter be able to function independently. This is useful for two reasons. First, it presents an environment where students can get used to working with Mini and JITed. Second, it provides a baseline
of behavior that students can fall back to if they are unsure how a case should be handled.

This is fulfilled with the separate classes and build targets for the JIT and the interpreter. The students can build an executable of the interpreter by itself and retain it for the rest of the project, as the interpreter itself is complete. In the case of a shared system, there could be a single pre-compiled executable that is shared with the students.

6.3 Remark on Portability

As a note, JITed should be fully portable to any target platform, so long as the LLVM library is available on that platform, and supports the target. ANTLR is not required for this task, as the generated C++ files are sufficient for handling the lexing and parsing. Thus, the requirements for the system to be compatible are that LLVM-10 libraries exist for the system it is being built on, and that LLVM support the target platform as a compile target.

6.3.1 Limitations

The most major limitation with the system is that compiling one function also compiles all functions that could be called during that initial function. For many small programs, this might mean that requesting compilation for the function the program is built around ends up compiling all the functions in the program. This is a consequence of how exceptions are thrown in the case of a missing symbol with the LLVM ORC LLJIT class. Not enough information is present at the point where control returns to the C++ context in order to reconstruct the invocation that caused the
exception. Without a significant redesign, likely using a different backend structure, this issue is likely unavoidable.
This chapter discusses potential expansions to JITed. Some sections in this document have alluded to locations where modifications or expansions might be made to the system. These expansions are discussed in further detail here, with a description of the general directions with potential for future work.

7.1 More precise JIT compilation

One of the primary limitations of JITed is the way that compilations will, in effect, cascade down into other functions. This is undesirable because it makes the performance impact of using a JIT unpredictable and potentially counterintuitive. For instance, compiling one function might compile most of the functions in a program, incurring a much higher up-front compilation cost than expected. Unless these functions would have been JIT compiled soon after anyways, this may result in a very different performance profile than would ordinarily be expected.

This limitation could be avoided in future work by opting for a more involved backend procedure. JITed used the LLVM ORC LLJIT class for this task which, while suited to the task of JIT’ing code, was intended for functions to always be compiled when run through it. There are therefore no provisions for partial compilation of the functions being run.

Should any expansions continue to use LLVM, then it may be required to make modifications to the LLVM library files in order to gain access to the necessary components.
7.2 Target language modifications

The most direct path for future work is to either modify the target language of Mini, or adapt the system to another language. Modifications to Mini may comprise extension to the type systems, such as adding arrays and strings, or the introduction of closures and further scoping rules. These each increase the complexity of the compiler required, but add deeper concepts to be explored. They also expand the potential for more interesting and computationally intensive programs, which add more opportunities for performance gains to become apparent.

Another option would be to fully change over to another target language, with all of the adjustments that entails. This opens a route for a system to be built from scratch, following in the direction of JITed, if not using the same codebase. This also allows for a different source language to be used for the system, although LLVM still provides likely the most mature and well-documented library for runtime compilation.

7.3 Deployment in a class

The ultimate objective of JITed is to be deployed in a classroom setting. Much of the work within the framework has been done to reduce the amount of prerequisite knowledge and development time for students in order to accommodate this. Future work could perform this deployment, as well as monitoring how effective the system is at teaching JIT concepts.

The evaluation of the system in the deployed setting could use a number of metrics, both in terms of student satisfaction with the system and also the students’ fluency in different concepts for both JITs and optimization more generally. Some ideas for lines of inquiry are presented below:
- How heavy did students feel the workload was?
- How comfortable were students with developing a compiler in general?
- How comfortable were students with optimizing a compiler?
- How confident were students in selecting appropriate optimizations for a given context?

These ideas could be expanded upon or become more precise, as they are only suggestions for the kinds of metrics that could be employed.
In this document, we have presented JITed, a framework for aiding in the task of teaching just-in-time compiler concepts in a classroom setting. JITed reduces the workload on the student by performing some of the ancillary tasks for the JIT to work, reducing the time required for a project to create a JIT so that it can fit within a quarter or semester term. This makes it feasible to teach JIT concepts either as the focus of an entire term, or time permitting a portion of the term. Because the target workload is very light, some of the support structures can be removed in order to have a more hands-on project or labs.

The light footprint makes JITed a reasonable first exposure to JIT compilers to students. Fully constructing a JIT compiler is likely outside the interests of most students, so for most this brief introduction to JITs is more than sufficient. Lectures can focus on the varied and interesting topics within JITs, with JITed providing some hands-on experience to compliment this.

We hope that JITed will be expanded upon by further frameworks in the future. We look forward to its use in any and all university settings to expose students to more details of how their tools work.
BIBLIOGRAPHY


APPENDICES

Appendix A

FULL MINI SPECIFICATION

Listed in this appendix are the informal semantics of Mini, as described by Aaron Keen. Some adjustments have been made, generally towards more concrete and restrictive semantics. These are due to the behavior of the interpreter being defined, and there being a need for consistency between the interpreter and compiler. Comments on the changes are italicized, and sections that are no longer entirely true are stricken through.

Semantics

The semantics for the language are given informally.

- Redeclarations of the same sort of identifier are not allowed, i.e., there cannot be two global variables with the same name, two formal parameters for a function with the same name, two variables local to a function with the same name, two functions with the same name, two structure declarations with the same name, or two fields within a structure with the same name.

- Local declarations and parameters may hide global declarations (and functions), but a local may not redeclare a parameter.

- Structure names are in a separate namespace from variables and functions.

- You may place functions and variables in separate namespaces. Or not. This is entirely up to you. Enjoy the freedom. They are in separate namespaces.
• Program execution begins in the function named `main` that takes no arguments and that returns an `int`. Every valid program must have such a function.

• The scope of each structure type is from the point of definition to the end of the file (this means that a structure type can only include elements of the primitive types and the structure types defined before it, though it must be allowed to include a member of its own type). You may extend the language to support file scope for structure declarations if you wish.

• The scope of each function is from the point of definition to the end of the file (though recursion must be supported, this restriction precludes mutually recursive functions). You may extend the language to support file scope for functions if you wish.

• The `if` and `while` statements have semantics equivalent to those of Java. They both require boolean guards.

• Assignment (strictly a statement) requires that the left-hand side and right-hand side have compatible types (equal in all cases except for `null`; `null` can be assigned to any structure type – boo for `null`).

• A declaration with a structure type declares a reference to a structure (the structure itself must be dynamically allocated).

• `null` may be assigned to any variable of structure type.

• The `. operator is used for field access (as in C and Java).

• All arguments are passed by value. For a structure reference, the reference itself is passed by value.
• **print** requires an integer argument and outputs the integer to standard out. A
  print with **endl** should print the integer followed immediately by a newline; a
  print without should print the integer followed immediately by a space.

• **read** reads (and evaluates to) an integer value from standard in.

• **new** dynamically allocates a new structure, but does not initialize it, and evalu-
  ates to a reference to the newly allocated structure. *Memory is sometimes, but
  not always initialized, depending on the context. This can be treated the same
  as uninitialized.*

• **delete** deallocates the referenced structure.

• Arithmetic and relational operators require integer operands.

• Equality operators require operands of integer or structure type. The operands
  must have matching type. Structure references are compared by address (i.e.,
  the references themselves are compared).

• Boolean operators require boolean operands.

• Boolean operators are **not** required to short-circuit (i.e., the user of this lan-
  guage should not assume short-circuiting). *The interpreter does not short-
  circuit booleans.*

• Each function with a non-void return type must return a valid value along all
  paths. Each function with a void return type must not return a value.