A Compiler for a Lisp-like Programming Language

Matthew Kosloski
Department of Computer Science
Purdue University Northwest
Hammond, Indiana
koslos@pnw.edu

***Abstract*—Algorithms that are employed to solve computational problems must be implemented in some high-level programming language. That is, a language that abstracts away the many low-level details of the computer (e.g., registers, memory addresses, the stack, calling conventions, etc.), thereby making it loosely coupled to the hardware. These high-level languages contrast greatly to low-level languages (e.g., assembly languages), which are very tightly coupled to a computer and its architecture. While this abstraction has facilitated accessibility and the speed at which programming can take place, it has simultaneously (by design) "shielded" programmers, many of whom take the abstraction for granted. There are several different ways of implementing a high-level programming language, one of which is through *compilation*. A *compiler* is a computer program, written in an implementation language, that translates a program written in a source language to an equivalent program written in a target language. In an attempt to better my understanding of the low-level details that high-level languages abstract away, I designed Torrey, a novel, high-level programming language, and implemented the language by developing a compiler. Torrey is "Lisp-like", meaning its syntax (form) and semantics (meaning) are inspired by that of the Lisp languages (e.g., Racket, Clojure, Scheme, Common Lisp, etc.). Moreover, the compiler translates Torrey programs (written in the high-level Torrey language) to equivalent x86-64 programs (written in the low-level assembly language). However, with the assistance of an intermediate language, the compiler can be modified to target additional languages other than x86-64.**

***Index Terms*—compilers, language implementation patterns, language translation, compiler construction, computer architecture, x86-64**

I. Introduction

Presently, in order to graduate from Purdue University Northwest (PNW) with a Bachelor of Science in computer science, students must take CS 420 Senior Project Design. This upper-level course provides students with the opportunity to work, either independently or in a group, towards writing sophisticated software. This spring 2021 semester, as a current CS 420 student, I decided to work independently towards building a compiler from scratch. Compilers are most definitely sophisticated pieces of software, and due to their opaque implementation, they are commonly thought of as being "scary" or "complex". As a matter of fact, the front cover of *Compilers: Principles, Techniques, and Tools*, commonly referred to as the "Dragon Book" by academics and compiler enthusiasts, depicts a literal knight slaying a purple, fire-breathing dragon.[1] Despite such connotations and the imagery depicted in the infamous Dragon Book, building a compiler (at least a simple one) need not be intimidating, for an incremental approach can be taken to reduce its complexity.

In the contents that follow, I will: communicate my prior knowledge and motivation for exploring compilers; define a compiler in terms of a black box and explain the significance of compilers in software engineering; introduce Torrey, a novel, high-level, Lisp-like programming language that I designed as input to the compiler[2]; provide a high-level technical description of the compiler that implements Torrey; describe the incremental approach adopted to tackle, or "slay", the complexity that inherently accompanies compiler construction; recount the challenges encountered while building the compiler; and conclude with retrospective remarks. To begin, I will provide some background information pertaining to my relevant experience with compilers and my motivation for exploring compilers as a senior project.

II. Motivation and Preliminary Work

Deciding to explore compiler construction for my senior project was a rather effortless decision for the following four reasons: a longing to learn more about the innerworkings of a compiler, which is one of many abstractions in a computer system; desire to bridge the gap between theoretical and practical application through the use of multidisciplinary knowledge; wanting to build upon my preliminary work with interpreters; and substituting for the lack of an undergraduate course in compilers. My "specialty" is in the designing and development of websites and web applications (I have had about half a dozen freelance gigs and two internships in the field). There are many layers of abstraction present while developing websites. For instance, when writing JavaScript code for the front-end of a web application, developers are "shielded" by the following abstraction layers: the APIs of front-end libraries and frameworks (e.g., React.js and Angular), built-in web browser APIs (e.g., ES6 fetch, DOM, geolocation, etc.), the JavaScript just-in-time compiler and

[1]The front cover art for the book *Compilers: Principles, Techniques, and Tools*, written by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, can be found here.

[2]I named the Torrey programming language after Torrey Pines State Natural Reserve, which is a state park in San Diego, California. It is a beautiful park off the coast of the Pacific Ocean. I visited the park in April 2015 (here is a photo of me at the park).

its implementation language (e.g., the famous JavaScript V8 engine written in C++), the browser process, the operating system on which the browser process runs, and the JavaScript programming language itself. I firmly believe that becoming acquainted with the innerworkings of some of these layers of abstraction makes me a more well-rounded, more informed software developer. By building a compiler from scratch, I can better appreciate the complexity of language translation units such as the aforementioned V8 JavaScript engine. In addition to my curiosity about the innerworkings of a compiler, I was also motivated by my desire to bridge the gap between theoretical and practical application through the use of multidisciplinary knowledge.

By constructing a compiler, I am bridging the gap between theory and application by using knowledge from many different computer science fields. As articulated in [1], "compiler writing touches upon programming languages, machine architecture, language theory, algorithms, and software engineering". Clearly, building a compiler is no small feat, for it requires a great deal of domain knowledge. In fact, while building the Torrey compiler, I spent many hours reading documentation to familiarize myself with a subset of the x86-64 (x86) instruction set. Although I took CS 223 Computer Architecture and Assembly Language, the assembly language used in that class was MIPS, which is much simpler than x86 assembly. In fact, the differences between MIPS and x86 assembly languages are so great that they are in two different categories: MIPS is classified as a reduced instruction set computer (RISC) architecture, whereas x86 is classified as a complex instruction set computer (CISC) architecture. In addition to possessing prior knowledge of computer architecture and assembly languages, compiler construction requires knowledge of language theory.

I learned concepts and techniques in CS 316 Programming Languages, and such knowledge has greatly facilitated in the design of the Torrey programming language and its implementation as a compiler. In the Spring 2020 semester, I took CS 316, and I became acquainted with the different language implementation methods (e.g., compilation, pure interpretation, hybrid, etc.) and examples of programming languages with such implementations, the differences between syntax and semantics of language, formal methods of describing syntax (e.g., the Backus-Naur Form metalanguage and context-free grammars), parse trees and abstract syntax trees, lexical analysis, syntax analysis, parsing techniques, and much more. In the course, I learned how to implement a basic lexical analyzer (which is one of the very first stages of a compiler) in lieu of building an actual interpreter or compiler. It was a phenomenal example of the practical application of a finite state machine, which is a rather theoretical and abstract concept. In addition to being motivated by curiosity about the innerworkings of a compiler and wanting to bridge the gap between theoretical and practical application through the use of multidisciplinary knowledge, I was also motivated by my preliminary work with interpreters.

The summer after taking CS 316, I applied my knowledge and constructed a simple interpreter for a superset of Torrey. In comparison to my compiler, I also adopted an incremental approach to the development of this simple interpreter.[3] As alluded to previously, *pure interpretation* is one of several language implementation methods. An **interpreter** is another way of implementing a programming language, an alternative to compilation, where the fetch-decode-execute cycle of a von Neumann machine is simulated using software. During the development of the interpreter, I consulted *Crafting Interpreters*, which is a textbook that walks the reader through a step-by-step process on how to build a tree-walk interpreter for a novel, C-style programming language called *Lox*.[4] Although I did not "finish" implementing the programming language, I was able to implement the following language features: binary arithmetic, lexically-scoped variables, booleans, strings, signed integers, signed real numbers, and control flow. Moreover, I was also motivated to build a compiler because of a lack of an undergraduate compilers course.

Unfortunately, as an undergraduate computer science student at PNW, there is no compilers course available to me. Although there is a graduate course in compilers, it focuses more on interpreters. Now that the reader is adequately informed on my background knowledge, my motivation to explore compiler construction, and my preliminary work with tree-walk interpreters, I will first define a compiler in terms of a black box and then subsequently explain the significance of compilers in software engineering.

III. About Compilers

A compiler, which at a high level is interpreted as a black box, exists to facilitate the programming of computers by humans. As shown in Fig. 1, a compiler is analogous to a black box: it takes an input, produces an output, and has an opaque implementation. However, formally, a **compiler** is a computer program written in an *implementation language* that translates a program written in a *source language* to an equivalent program in a *target language*. It is very important to be cognizant of the roles of such languages.

Fig. 1. At the highest level, a compiler is interpreted as being a black box with an input, and output, and an opaque implementation.

The *implementation language* is the programming language in which the actual compiler is written (the implementation

[3]The source code of the final iteration of the interpreter, humorously named *Truncated Tarantala*, can be found here.

[4]A free, digital copy of *Crafting Interpreters*, written by author Bob Nystrom, can be found here.

languages of the Torrey compiler are Java and C), the *source language* is the programming language in which the input program is written (the Torrey compiler takes programs written in, well, Torrey), and the *target language* is the programming language in which the output program is written (by default, the Torrey compiler outputs x86 assembly programs but can easily be modified to output programs in other languages). Also, it is important to note that the output program must be *equivalent* to the input program. That is, the ouput program should do *exactly* what the input program instructs it to do. Naturally, the reader may ask why even go through the trouble of translating a computer program from one language to another. The answer is simple and it is why compilers are significant: digital computers do not understand programming languages.

Compilers are significant pieces of software that improve the speed at which computers can be programmed, for they translate computer programs written in high-level programming languages, which are human-friendly, to low-level assembly languages, which are CPU-friendly. In the 1940s, digital computers were programmed using **binary code**, or *machine code*, which is a sequence of 0s and 1s representing low and high electrical signals. Although computers can only understand binary code, programming in binary code was incredibly tedious, so a few decades later, assembly languages were invented. Assembly languages are an improvement over programming in machine code because they use short words and symbols rather than just 0s and 1s. However, assembly language programs are linear in structure and thus lack depth and scope. Moreover, assembly languages are tightly coupled to an individual CPU and its architecture; generally every CPU has a different assembly language (e.g., Intel CPUs, such as the i7-9700K, execute programs written in x86 assembly whereas the new Apple M1 CPU executes programs written in ARM assembly). Due to such shortcomings, beginning in the 1950s, high-level programming languages began to appear.

Due to the aforementioned disadvantages to assembly programming, high-level programming languages such as C, Java, Python, and Torrey were invented. As articulated in [2], Fortran, which first appeared in the mid-1950s, is considered the first compiled high-level programming language. Clearly, programming digital computers using high-level programming languages is very desirable, for even today they are used to write programs. The relationship between binary code, assembly code, and high-level code is depicted in Fig. 2. From the figure, it can be inferred that as the abstraction level increases, so does the intuition level, and that is exactly why compilers exist. Technically, they do not need to exist at all, but who would want to write programs in binary (part (c) in Fig. 2) or assembly language (part (b) in Fig. 2)? Moreover, who would want to rewrite their program in a different assembly language for every CPU they wanted to target? Compilers exist merely as a convenience, but it turns out that they are so *incredibly* convenient that their existence is almost necessary. Now that the reader is familiar with what a compiler is and the role that they play in software engineering, I can continue by introducing the syntax and semantics of the Torrey programming language.

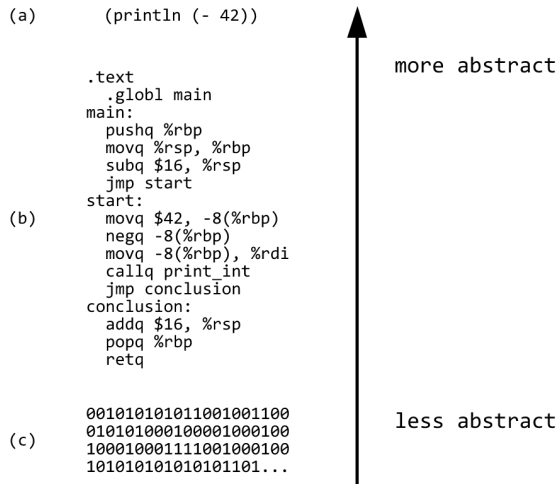

Fig. 2. High-level programming language (a) code is more abstract than assembly language code (b), which is more abstract than binary code (c).

IV. The Torrey Programming Language

A. The Syntax of Torrey

The syntax, or form, of syntactically valid Torrey programs can be formally described by the context-free grammar in Listing 1, which is written in the Extended Backus-Naur Form, or EBNF, metalanguage.[5] A **grammar** is a collection of one or more *rules* that are used to generate sentences in a language. In the context of this paper, the language is the Torrey programming language and the sentences are Torrey programs.

A **rule** has a left-hand side, or LHS, and a right-hand side, or RHS, and is of the form `LHS -> RHS` (read as "LHS expands to RHS"). Rules can have exactly one RHS (e.g., `int -> { 0-9 }+`) or more than one RHS (e.g., `prim -> int | bool`) delimited by a `|` (pipe) symbol that reads as "or". A **language** is a set of zero or more sentences, and a **sentence** is a sequence of zero or more *terminal* symbols. In addition to terminal symbols, there are also *non-terminal* symbols that can "expand to" or be "replaced with" a RHS of that non-terminal. In the grammar shown in Listing 1, terminals are surrounded in single quotation marks and non-terminals are in normal font. Further, there are different types of grammars such as regular, context-sensitive, and context-free; however, only context-free grammars are relevant to this discussion.

[5]According to Merriam-Webster's dictionary, a **metalanguage** ia a language used to talk about language.

```
program -> { expr }*

expr    -> prim | id | unary | binary
         | print | let | not
         | and | or | if

prim    -> int | bool
bool    -> 'true' | 'false'
int     -> { 0-9 }+
id      -> { a-zA-Z_$ }+ { a-zA-Z0-9_$!?- }*
unary   -> '(' '-' expr ')'
binary  -> '(' ('+' | '-' | '*' | '/' |
              '==' | '<' | '<=' | '>' | '>=')
              expr expr ')'
print   -> '(' ('print' | 'println')
              { expr }+ ')'
let     -> '(' 'let' '[' { id expr }* ']'
              { expr }* ')'
not     -> '(' 'not' expr ')'
and     -> '(' 'and' expr { expr }+ ')'
or      -> '(' 'or' expr { expr }+ ')'
if      -> '(' 'if' expr expr [ expr ] ')'
```

Listing 1. The formal syntax of the Torrey programming language described by a context-free grammar written in the EBNF metalanguage.

A **context-free grammar**, or CFG, is a grammar in which the LHS of every rule is a single non-terminal symbol. In [3], Elaine Rich, the author of the textbook used in PNW's automata course (CS 410), explains that such grammars are called "context-free" because "...the decision to replace a nonterminal by some other sequence is made without looking at the context in which the nonterminal occurs." This excerpt can be better understood with an example *derivation*.

A **derivation** is a sequence of grammar rule applications and can be used to prove that a particular sentence is in a language. Here, I define a **rule application** as a step in a derivation wherein the leftmost non-terminal is replaced with exactly one of its RHSs. Consider the example derivation of `(- 42)` shown in Listing 2. In line 1, the non-terminal symbol `program` is the *start symbol* and `{ expr }*` (read as "zero or more instances of `expr`") is the RHS of `program` in the grammar shown in Listing 1. A **start symbol** is a non-terminal symbol from which every sentence in a language can be derived. Thus, all derivations start with the same non-terminal symbol. Put another way, every Torrey program can be derived by starting from the non-terminal symbol (start symbol) `program`. In line 2, the leftmost (and only) non-terminal `expr` has been replaced with `unary`, which is one of the many RHSs of `expr`. Line 3 can be derived from line 2 by replacing the non-terminal `unary` with its RHS `'(' '-' expr ')'`. Line 4 can be derived from line 3 by replacing the leftmost non-terminal `expr` with `prim`. Line 5 can be derived from line 4 by replacing the leftmost non-terminal `prim` with `int`. Finally, in line 6, the program `(- 42)` can be derived from line 5 by replacing the non-terminal `int` with a terminal symbol `'42'`. Now, I will return to the aforementioned excerpt by Elaine Rich.

The grammar in Listing 1 is context-free because every LHS is a single non-terminal symbol and, when performing derivations, the decision to replace a non-terminal with a RHS is made without looking at context. For instance, in Listing 2, non-terminal `int` is replaced with terminal `'42'` without considering the symbols surrounding `int` in line 5. However, suppose I were to modify the grammar in Listing 1 by removing the rule `int -> { 0-9 }+` and replacing it with rules `-int -> { 0-9 }` (reads as "-int expands to exactly one of 0-9") and `+int -> { 0-9 }*` (reads as "+int expands to zero or more of 0-9"). After this change, the grammar in Listing 1 would no longer be context-free. This is because the additional rules have terminal symbols (`-` and `+`) in their RHSs. Moreover, in line 5 of the derivation, the RHS with which non-terminal `int` is replaced is dependent upon whether non-terminal `int` is prefixed with a '-' or '+' terminal. Context-free languages lack such determinism. Now that the reader is sufficiently informed on grammars, context-free grammars, and how to derive sentences from grammars, I will continue by summarizing the rules of the Torrey grammar.

```
program => { expr }*                  1
        => unary                      2
        => '(' '-' expr ')'           3
        => '(' '-' prim ')'           4
        => '(' '-' int ')'            5
        => '(' '-' '42' ')'           6
```

Listing 2. A derivation of the Torrey program `(- 42)`.

The Torrey grammar shown in Listing 1 can be summarized as follows (each itemized bullet corresponds to an individual rule in the grammar):

- A Torrey program is a collection of zero or more expressions.
- An expression can be a `prim` (short for *primitive*), `id` (short for *identifier*), unary, binary, print, let, not, and, or, or if.
- There are two primitive expressions: integers (`int`) and booleans (`bool`).
- A boolean expression evaluates to either `true` or `false`.
- An integer is a sequence of one or more numbers in the range 0-9.
- An identifier must start with either a lowercase letter (`a-z`), an uppercase letter (`A-Z`), an underscore (`_`), or a dollar sign (`$`) and may be followed by any combination of lowercase letters, uppercase letters, underscores, dollar signs, exclamatory signs (`!`), question marks (`?`), or dashes (`-`).
- A unary expression is fully-parenthesized and contains a `-` symbol followed by an expression.
- A binary expression is fully-parenthesized and contains either a `+`, `-`, `*`, `/`, `==`, `<`, `<=`, `>`, or `>=` symbol followed by two expressions.
- A `print` expression is fully-parenthesized and contains either the `print` or `println` reserved word, followed by

one or more expressions.
- A `let` expression is fully-parenthesized and contains the `let` reserved word, followed by an opening bracket `[`, followed by zero or more pairs of an identifier and expression, followed by a closing bracket `]`, followed by zero or more expressions.
- A `not` expression is fully-parenthesized and contains the reserved word `not` followed by an expression.
- An `and` expression is fully-parenthesized and contains the reserved word `and` followed by two or more expressions.
- An `or` expression is fully-parenthesized and contains the reserved word `or` followed by two or more expressions.
- An `if` expression is fully-parenthesized and contains the `if` reserved word, followed by two expressions, and an optional third expression.

B. The Semantics of Torrey

A Torrey program is *semantically* valid if and only if it conforms to the following rules:

- Expressions are evaluated from the inside out. For example, in `(* 1 (+ 2 (- 3)))`, `(- 3)` is evaluated first, then `(+ 2 (- 3))`, and finally `(* 1 (+ 2 (- 3)))`. Thus, the order of operations is automatic.
- Expressions are executed sequentially. For example, in `(print 42) (print (- 42))`, `(print 42)` is evaluated and executed before `(print (- 42))`.
- An identifier (`id`) is a name that evaluates to an expression.
- A unary expression containing the `-` terminal symbol evaluates to the negation of its integer operand. The operand must evaluate to an integer.
- A binary expression containing the `+` terminal symbol evaluates to the sum of its two integer operands. The operands must evaluate to integers.
- A binary expression containing the `-` terminal symbol evaluates to the difference of its two integer operands. The operands must evaluate to integers.
- A binary expression containing the `*` terminal symbol evaluates to the product of its two integer operands. The operands must evaluate to integers.
- A binary expression containing the `/` terminal symbol evaluates to the quotient of its two integer operands. The operands must evaluate to integers and the second operand must not evaluate to zero (e.g., `(/ 42 0)`).
- A binary expression containing the `==` terminal symbol evalutes to a boolean. The two operands must evaluate to integers or booleans. If the evaluation type of one operand is not the same as the other (e.g., `(== true 42)`), then an error is thrown.
- A binary expression containing the `<` terminal symbol takes two operands that evaluate to integers. The binary expression evaluates to `true` if the first operand is less than the second operand (e.g., `(< 60 420)`) and evaluates to `false` otherwise (e.g., `(> 420 69)`).
- A binary expression containing the `<=` terminal symbol takes two operands that evaluate to integers. The binary expression evaluates to `true` if the first operand is less than or equal to the second operand (e.g., `(<= 420 420)`) and evaluates to `false` otherwise (e.g., `(>= 421 420)`).
- A binary expression containing the `>` terminal symbol takes two operands that evaluate to integers. The binary expression evaluates to `true` if the first operand is greater than the second operand (e.g., `(> 420 22)`) and evaluates to `false` otherwise (e.g., `(> 69 420)`).
- A binary expression containing the `>=` terminal symbol takes two operands that evaluate to integers. The binary expression evaluates to `true` if the first operand is greater than the second operand (e.g., `(>= 999 999)`) and evaluates to `false` otherwise (e.g., `(>= (- 68) 69)`).
- A `print` expression evaluates one or more expressions, sending their values to standard output. The operands need not be of the same type (e.g., `(print true (- 1))`.
- A `let` expression binds zero or more expressions to identifiers within a lexical scope. Within the lexical scope, there can be zero or more expressions. Thus, `(let [])` is both syntactically and semantically valid. If there is at least one expression in the lexical scope, then the `let` expression evaluates to the value of the last expression in the scope. For example, `(let [a 5] (+ a a) (* a a))` evaluates to `(* a a)`, or 25. If there are no expressions in the lexical scope (e.g., `(let [])`), then the `let` expression evaluates to `undefined`. When scopes are nested, the binding closest to the identifier is used. For example, `(let [a 1] (let [a 2] a))` evaluates to `2`. Identifiers can be bounded to identifiers within scope (e.g., `(println (let [a 4 b 5] (let [c (+ a b) d (* 2 c)] (* c d))))`).
- A `not` expression takes a boolean as an operand and evaluates to `false` if the operand evaluated to `true` and `true` if the operand evaluated to `false`.
- An `and` expression evaluates two or more boolean expressions and returns `true` if and only if each expression is truthy and returns `false` otherwise.
- An `or` expression evaluates two or more boolean expressions and returns `true` if at least one expression is truthy and returns `false` otherwise.
- An `if` expression controls the flow of a program. If the first operand (the test condition) is true, then the `if` expression evaluates to the second operand (the consequent). Otherwise, the `if` expression evaluates to the third operand (the alternative) or `undefined` if there is no alternative.

Finally, it should be noted that syntactic validity does not imply semantic validity. In other words, just because a Torrey program is *syntactically* valid (that is, using a derivation, it can be proved that a Torrey program is in the Torrey language), that does not mean that the program is *semantically* valid (that is, the program conforms to all the semantic rules outlined in the aforementioned section). For example, the following Torrey programs are syntactically valid but are *not* semantically valid

and thus will not compile:

- `(+ 2 (print 3))`
- `(println (+ 10 (if false 32 (== false true))))`
- `(let [foo 42] (if (let [bar foo] bar) (print foo) (print (- foo))))`
- `(println (let [x 42] (/ x (if (< x 3) 1 0))))`

Determining exactly why these programs do not compile can be an exercise for the reader.

Now that the reader is familiar with the design of the Torrey programming language (its syntax and semantics), I can continue to describe the language's *implementation*. As alluded to previously, a programming language can be *implemented* in a number of different ways, through either compilation (example languages include C and C++), pure interpretation (example languages include Python), or a more hybrid approach (example languages include Java). I chose to implement the programming language through compilation by constructing a compiler from the ground up (that is, each *phase*, or *pass*, of the compiler has been written by me). In the contents that follow, the reader will first learn about the compiler software itself: the location of its version control repository, installation, dependencies, and command-line interface (CLI). After this, the reader will be provided with a high-level technical description of the compiler that implements Torrey.

V. The Torrey Compiler

The Torrey compiler is implemented in Java and C and, as shown in Fig. 3, has two parts: a *front-end* and a *back-end*. However, before I delve deeper into such an implementation, I will first provide information on the compiler software itself, starting with its version control system.

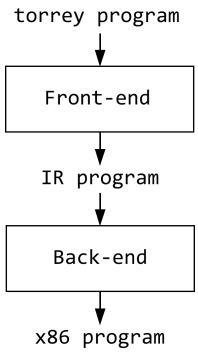

Fig. 3. At a high level, the Torrey compiler has a front-end and a back-end.

A. Version Control

I decided to use Git version control to faciliate the development of the compiler. **Version control** is software used to track the status and changes of every file within a repository, or repo. The compiler repo is hosted on GitHub and can be found here. In the next section, I identify the software that is required to use the compiler and to run the produced executables.

B. Software Dependencies

In order to use the compiler, the reader must have the JDK (Java Development Kit) installed on their computer and `java` must be in their `PATH`. Additionally, the reader must have `gcc` installed and in their `PATH`, which is the GNU Compiler Collection. This collection of software is necessary to build the runtime object code and to assemble and link the object code with the compiled Torrey program to produce an executable. Finally, the reader must be on a Linux machine, for the outputted x86 code is for Linux. The compiler has been tested using the following dependency versions:

- `java` openjdk 11.0.11
- `gcc` 7.5.0
- Ubuntu 18.04 LTS

If the reader cannot fulfill these requirements, there is another option: containerization. If the reader does not want to pursue containerization, then they can skip to the section titled *C. Download the Compiler and Runtime from GitHub*.

If the reader does not want to, or cannot, install the aforementioned dependencies, then they can run the compiler within a Docker *container* using Docker Desktop. A **container**, which is a running instance of an *image*, is an independent piece of software that packages up all the necessary dependencies, allowing software to run more predictably within an isolated environment.[6] Containers have everything that is needed to run an application, from the software itself, along with its dependencies, a file system, and even the OS on which it runs. Docker Desktop, which is used to run containers, can be downloaded from here.

Once the reader has Docker Desktop installed, they can *pull*, or download, the Docker image from which individual containers can be instantiated. Then, to use the compiler, simply start up an individual container and pass it your program as an environment variable. The following summarizes the steps necessary to run the compiler within a container without installing anything other than Docker Desktop:

1) Verify the installation of Docker Desktop by opening a shell and typing `docker`.
2) Download, or *pull*, the Docker image from the offical Docker image repository: `docker pull matthewkosloski/torrey-playground`.
3) Verify that the image has been downloaded by listing the currently installed images: `docker image ls`.
4) Run the Torrey compiler by spinning up an individual container like so:
   `docker run --rm --env TORREY_PROGRAM="(println 42)" matthewkosloski/torrey-playground`. If all goes well, the reader should see `42` on the screen.

[6]Technically, software running within a Docker container is not completely isolated from the host OS as it is possible for containers to run as `root` user. However, there are security precautions that can be taken to make sure containers running on the host OS run as non-root.

If the reader pursued containerization, then they can skip to the section titled *D. Command Line Interface*. Else, the reader can continue onto the next section wherein I discuss the downloading of the compiler.

C. Download the Compiler and Runtime from GitHub

To download the latest release of the compiler and its runtime, visit this location within the repo and download the asset named `torreyc-x.x.x.zip`, where `x.x.x` is the latest semantic version number of the compiler. Once the `.zip` file is downloaded, please extract it. Upon extracting, the reader will be greeted with a folder containing three files:

- `torreyc-x.x.x.jar`. The Torrey compiler itself in the form of a Java executable (`.jar` file) that is to later be run on the Java Virtual Machine, or JVM.
- `runtime.c`. The Torrey runtime implementation. The aforementioned compiler automatically builds this into *object code*, which is subsequently assembled and linked with the compiled Torrey program to produce an executable file.
- `runtime.h`. The Torrey runtime header information.

Now that the reader has installed the necessary software (`java`, `gcc`, and Linux 18.04) and downloaded the latest compiler and runtime, they can begin familiarizing themselves with the compiler's command-line interface.

D. Command Line Interface

In order to use the Torrey compiler, users must interact with its command-line interface, or CLI. CLIs contrast greatly to the traditional graphical user interfaces, or GUIs, wherein users provide input to applications through interactions with interface elements such as text boxes, buttons, menus, etc. To give input to a program with a CLI, users must type arguments, or *flags*. To view the documentation for the compiler's CLI, simply run the compiler without any input like so: `java -jar torreyc-x.x.x.jar`. I will demonstrate a subset of the CLI, namely input and output.

Input. To give the compiler input, run the compiler with the `-i` argument. Suppose there is a file named `foo.torrey` with the following contents:

```
; foo.torrey
(let [foo (+ 2 3) bar (let [baz 9] (* foo baz))]
  (println foo bar))
```

The above `foo.torrey` program can be compiled by running: `java -jar torreyc-x.x.x.jar -i foo.torrey`. This will produce an executable named `a.out` within the current directory. In addition to the aforementioned `-i` argument, the compiler can also accept input from standard input like so: `echo "(print 1)" | java -jar torreyc-x.x.x.jar`.

Output. By default, the compiler: (1) compiles the input Torrey program into an equivalent 64-bit x86 assembly code program, (2) assembles and links the assembly with the dependent runtime object code, and (3) builds an executable named `a.out` in the current directory. To override this default behavior, supply the compiler with exactly one of the command-line arguments listed in Table I.

Flag	Description
`-o <filename>`	Place the output of the compiler into a file called `<filename>` in the current directory.
`-L`	Lex only; do not parse, generate intermediate code (IR), compile, or assemble.
`-p`	Parse only; do not generate IR, compile, or assemble.
`-ir`	Generate IR only; do not compile or assemble.
`-S`	Compile only; do not assemble.

TABLE I
COMMAND-LINE ARGUMENT FLAGS TO OVERRIDE OUTPUT BEHAVIOR

It should be noted that only one of the flags shown in Table I can be supplied, and the output can go to either the standard output stream or the file system. If the `-o` flag is present, then compiler output is written to the file system. Else, compiler output is printed to standard output.

For example, to stop the compiler after the lexical analysis, or lexing, stage (that is, refrain from parsing, generating intermediate code, compiling, and assembling) and send the output of lexical analysis to the standard output stream, simply provide the `-L` flag without the `-o` flag like so: `echo "(println (-1))" | java -jar -L`. After running the command, the following will be printed to standard output:

```
<'(',LPAREN,1:1,1:2>
<'println',PRINTLN,1:2,1:9>
<'(',LPAREN,1:10,1:11>
<'-',MINUS,1:11,1:12>
<'1',INTEGER,1:12,1:13>
<')',RPAREN,1:13,1:14>
<')',RPAREN,1:14,1:15>
<'',EOF,-1:-1,-1:-1>
```

The compiler can also be stopped after the syntax analysis, or parsing, stage (that is, refrain from generating intermediate code, compiling, and assembling). To stop the compiler after parsing and send the output to a file named `parse.json`, run: `echo "(println (-1))" | java -jar torreyc-x.x.x.jar -p -o parse.json`. After running the command, the following will be written to a file named `parse.json` in the current directory (partly truncated for brevity):

```
{"children": [{
    "evalType": "UNDEFINED",
    "node_type": "PrintExpr",
    "children": [{
      "evalType": "INTEGER",
      "node_type": "UnaryExpr",
      "children": [{
        "evalType": "INTEGER",
        "node_type": "IntegerExpr",
        "token": {
          "rawText": "1",
          "endIndex": 12,
          "beginIndex": 11,
          "type": "INTEGER",
          "beginLineIndex": 0,
          "startPos": {"col": 12, "line": 1},
          "endPos": {"col": 13, "line": 1}
        }
      }],
      "token": { ... }
    }],
    "token": { ... }
  }],"token": "null"}
```

In addition to being stopped after lexing and parsing, the compiler can also be stopped after the intermediate code generation, or IR gen, stage (that is, refrain from compiling and assembling). To stop the compiler after IR generation and send the output of the IR gen stage to the standard output stream, simply provide the `-ir` flag without the `-o` flag like so: `echo "(println (-1))" | java -jar torreyc-x.x.x.jar -ir`. After running the command, the following will be printed to standard output:

```
t0 = - 1
param t0
call println, 1
```

Lastly, the compiler can also be stopped after compilation (that is, refrain from assembling). To stop the compiler after compiling and send the output of the compilation stage to the standard output stream, simply provide the `-S` flag without the `-o` flag like so: `echo "(println (-1))" | java -jar torreyc-x.x.x.jar -S`. After running the command, the following will be printed to standard output:

```
.text
  .globl main
main:
  pushq %rbp
  movq %rsp, %rbp
  subq $16, %rsp
  jmp start
start:
  movq $1, -8(%rbp)
  negq -8(%rbp)
  movq -8(%rbp), %rdi
  callq print_int
  jmp conclusion
conclusion:
  addq $16, %rsp
  popq %rbp
  retq
```

Now that the reader is familiar with the compiler's CLI, specifically the input and output mechanisms, I will provide a high-level technical description of the Torrey compiler, reviewing both the compiler's *front-end* and *back-end*. A compiler **front-end** performs lexical analysis, syntax analysis, semantic analysis, and intermediate code generation. In contrast, a compiler **back-end** translates the intermediate code to a target language. This discussion begins with an explanation of why the front-end and back-end are separated.

As articulated in [4], if you had M different source languages and N different target languages, then, in principle, this would require $M \cdot N$ individual compilers. Luckily, this monumental task can be greatly reduced with the introduction of an intermediate representation, or IR. The IR acts as a middle-man between the high-level source language and the low-level target language. With the addition of an IR, only M front-ends and N back-ends are required, resulting in a total of only $M + N$ individual compilers. It should be noted that in the context of the Torrey compiler, there is only $M = 1$ source languages (that is, the Torrey programming language). However, when designing the compiler, I wanted it to be very easy to add additional target languages, or back-ends. Thus, although there is currently only one back-end (that is, the x86 back-end for PC Linux), I consider there to be N *potential* back-ends (as shown in Fig. 4). The keen reader may have recognized that, with $M = 1$, $M \cdot N$ and $M + N$ are essentially the same and thus may start to question the purpose of the IR in the Torrey compiler. However, I firmly believe that an IR serves a purpose, even when $M = 1$.

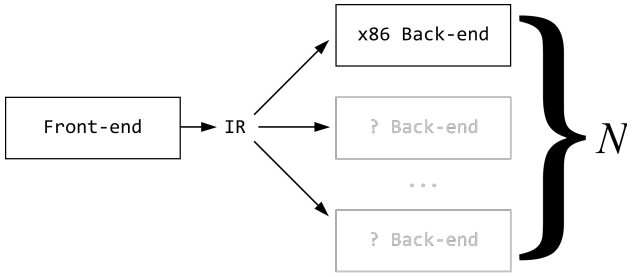

Fig. 4. The Torrey compiler has $M = 1$ front-ends and an x86 back-end, but because more back-ends can easily be integrated later on, there are N *potential* back-ends.

Although $M = 1$ for the Torrey compiler, there are still several benefits to implementing an IR to separate front-end activities from back-end activities. For one, the front-end and back-end are very loosely-coupled, which facilitates open source collaboration. For instance, one developer can be responsible for a single front-end that produces a "rock-solid" IR, and another developer can be responsible for a single back-end targeting a very specific computer target (down to the OS, vendor, and CPU architecture). The back-end developer need not know about the front-end activities (lexical analysis, syntax analysis, semantic analysis, IR code gen) and the front-end

developer need not be familiar with any assembly language. Not only does the front-end and back-end separation improve collaboration, it is, in general, a good software engineering practice, namely separation of concerns. Additionally, with the introduction of an IR, you only have to perform the aforementioned front-end activities once. Moreover, if I ever wanted to increment M in the compiler to support backwards compatibility (have the compiler support older versions of the source language), then only one back-end would be needed. Now that the reader is informed of the significance of an IR to separate front-end and back-end compiler activities, I will continue by providing a high-level description of the compiler front-end and back-end, starting off with the front-end.

E. Compiler Front-end

As stated previously and shown in Fig. 5, the front-end is the first part of the Torrey compiler and performs lexical analysis, syntax analysis, semantic analysis, and intermediate code generation.

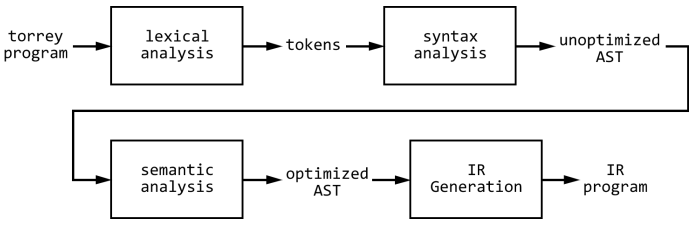

Fig. 5. The front-end activities of the compiler, showing the input and output of each activity.

1) Lexical Analysis: The **lexical analysis**, or lexing, stage of the compiler's front-end takes in the input Torrey program as a string and outputs a collection of *tokens*. A **token** is a unit of output from the lexer and holds attributes to facilitate parsing and the creation of error messages. Such attributes include

- the *type* of token (e.g., `LPAREN` for the `(` terminal symbol, `PLUS` for the `+` terminal symbol, `PRINTLN` for the `println` reserved word, etc.),
- the *raw text* from which the token was derived (a sequence of zero or more terminal symbols), and
- the *line number* and *column number* at which the raw text starts and ends in the input string.

Tokens are created by looping linearly through the input string and keeping track of a cursor, which holds the index of the currently examined character of the string. In each iteration of the loop, a `nextToken()` method is invoked, which appends a new token to the accumulated collection of tokens. As shown in Listing 3, the `nextToken()` method is implemented using a `switch` statement that switches on the character under the cursor. The method examines the current character under the cursor and produces a token based on a pattern match. Since the entire input string is known before lexical analysis begins, the lexer can perform *character lookahead*.

```
nextToken() {                                                    1
  currentChar = input[cursor++]                                  2
  switch (currentChar) {                                         3
    case '(': addToken(LPAREN)                                   4
    // ...                                                       5
    case '=':                                                    6
      if (match('=')) addToken(EQUAL_EQUAL)                      7
   // ...                                                       8
  }                                                              9
}                                                                10
```

Listing 3. Pseudocode implementation of the `nextToken()` procedure.

A **character lookahead** occurs when the lexer examines the kth character to the *right* of the cursor. An example of character lookahead is shown on line 7 of Listing 3. In line 7, the lexer is performing a pattern match for a `==` terminal symbol. However, just knowing the current character is not enough to match for `==`. Thus, to match for `==`, the lexer calls `match()`, which returns `true` if the character to the right of the cursor is `=`. After the lexer finishes looping through the input string, a final *end-of-file*, or EOF, token is appended to the end of the token collection. Finally, this collection of tokens is passed to the syntax analyzer, which is the subsequent compiler stage.

2) Syntax Analysis: The **syntax analysis**, or parsing, stage of the compiler's front-end takes in a collection of tokens and outputs an *abstract syntax tree*. An **abstract syntax tree**, or AST, is a tree data structure that captures the relationship among operators and operands of a program. For example, the AST of the Torrey program `(println 1 (+ 2 3) (- (* 4 5)))` is shown in Fig. 6. During parsing, a *parser* produces an AST by looping linearly through the input token collection and keeping track of a cursor, which holds the index of the currently examined token within the collection. In each iteration of the loop, a new node is created and added to the AST.

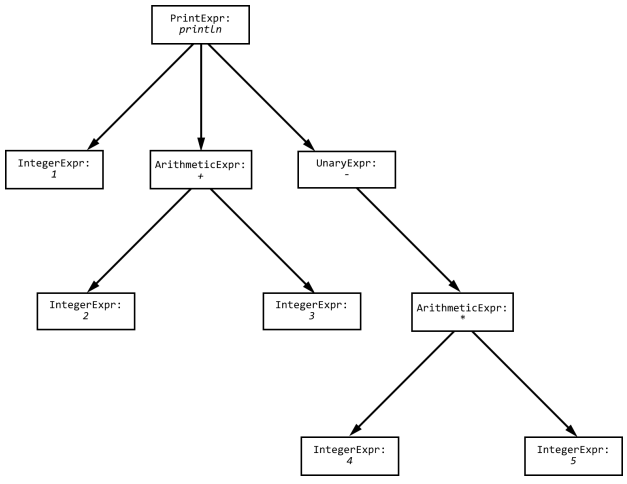

Fig. 6. The unoptimized AST for the Torrey program `(println 1 (+ 2 3) (- (* 4 5)))`.

A **parser** is an implementation of the grammar of the language being parsed. That is, for every rule `LHS -> RHS` in the grammar, there is a method `LHS()` in the parser that returns an AST node depending on `RHS`.[7] For example, consider the rule `program -> { expr }*` in the Torrey grammar shown in Fig. 1. This rule can be read as, "a `program` AST node has zero or more `expr` AST nodes." The implementation of this grammar rule is shown in Listing 4. Notice on line 4 in Listing 4 that there is a call to an `expr()` method. This is because, within the Torrey grammar shown in Fig. 1, there is the following rule: `expr -> prim | id | unary | binary | ... | if`. The implementation of this grammar rule is shown in Listing 5.

The grammar rule that is implemented in Listing 5 has many RHSs, or alternatives, and is of the form `LHS -> alt1 | alt2 | ... | altN`. As demonstrated in Listing 5, such grammar rules can be implemented using control flow. The method `peek()` indicates whether the type of the first token of lookahead (that is, the token under the cursor) is one of the specified token types. For example, line 11 in Listing 5 checks to see if the first token of lookahead is of type `TRUE`, `FALSE`, or `INTEGER`. If `peek()` returns `true`, then a `prim`, or primitive, expression has been encountered and we want to return a `prim` AST node via `prim()`. Notice how this implementation is almost identical to the grammar rule shown in Listing 1: `prim -> int | bool`. Similar to how the lexer uses a second *character* of lookahead, the parser uses a second *token* of lookahead.

```
1 program() {
2   exprs = []
3   while (hasTokens()) {
4     exprs.push(expr())
5   }
6   return Node(``program'', exprs)
7 }
```

Listing 4. A pseudocode implementation of the `program -> { expr }*` grammar rule found in the Torrey grammar shown in Listing 1.

The parser uses a second token of lookahead to differentiate between expressions that start with the same tokens. For example, in Torrey, expressions `(if true 42)` and `(+ 2 3)` both begin with an `LPAREN` (`(`) token. Thus, the parser uses a `peekNext()` method, which indicates whether the type of the token to the right of the cursor (that is, the second token of lookahead) is one of the specified token types. For example, in line 3 of Listing 5, the parser uses `peekNext()` to check whether the token after the `LPAREN` token is a binary operator (one of `PLUS`, `STAR`, `SLASH`, `EQUAL`, `LT`, `LTE`, `GT`, or `GTE`). If `peekNext()` returns `true`, then a binary expression has been encountered and we want to return a binary AST node via `binary()`. Finally, if execution reaches line 15 in Listing 5, then an error message is displayed to the user and an empty AST node is returned, indicating a parse error. Once every token has been examined by the parser and the entire AST has been produced, then the parser returns the root AST node, which is the given to the semantic analyzer.

[7]The actual implementation of the Torrey context-free grammar can be found here on GitHub.

```
expr() {                                          1
  if (peek(LPAREN)) {                             2
    if (peekNext(PLUS, STAR, ..., GTE))           3
      return binary()                             4
    else if (peekNext(MINUS))                     5
      return binaryOrUnary()                      6
    // ...                                        7
    else if (peekNext(IF))                        8
      return if()                                 9
  }                                               10
  else if (peek(TRUE, FALSE, INTEGER))            11
    return prim()                                 12
  else if (peek(ID))                              13
    return id()                                   14
  error(``Expected an expression'')               15
  return Node()                                   16
}                                                 17
```

Listing 5. A partial pseudocode implementation of the `expr -> prim | id | unary | binary | ... | if` grammar rule found in the Torrey grammar shown in Listing 1.

3) Semantic Analysis: During **semantic analysis**, the compiler performs three *passes* over the AST to obtain information to better inform subsequent compiler stages. Within each **pass**, the compiler visits every node in the tree and modifies a part of the AST. The AST is passed by reference, and thus the input to each pass is the output from the previous pass. The compiler performs the following three passes, in order:

1) The first semantic compiler pass creates the *lexical environments*. Within this pass, every `LetExpr` AST node (that is, the node that represents `let` expressions) is visited and decorated with a lexical environment. A **lexical environment** is composed of a *symbol table* and a reference to the enclosing parent environment (if no parent, then the reference is `null`). A **symbol table** is implemented as a hash map and maps identifiers to *symbols*. A **symbol** holds the following properties about an identifier: `name` (the name of the identifier), `type` (the data type of the expression bounded to the identifier), `category` (the category of identifier), `expr` (a reference to the expression that is bounded to the identifier), and `address` (used by subsequent compiler stages). Lexical environments and scoping can be demonstrated with the example program shown in Listing 6.

```
(let [a 1 b (- 2)]
       (println a b)
       (let [a 3]
            (println a b)))
```

Listing 6. A nested `let` expression.

The lexical environment that is visible to the expressions within the body of the inner `let` expression is shown

in Fig. 7. In the aforementioned figure, `Env@63` is associated with the inner `let` expression's AST node, whereas `Env@45` is associated with the outer `let` expression's AST node. As shown in the figure, these two environments form a linked list. References to identifiers within a `let` body are realized by traversing this linked list, starting with the local environment. For example, the `println` expression in the body of the inner `let` expression references identifiers `a` and `b`. Such identifiers are realized by first searching for them within `Env@63`, which is the local environment. From within this environment, `a` is found and is thus bounded to `IntegerExpr@84: 3`. However, `b` is not found within this environment, so the search continues by following the reference to the parent environment, `Env@45`. From within this environment, `b` is found and thus is bounded to `UnaryExpr@106: (- 2)`. If, however, `b` were *not* found within this parent environment, then a semantic error would be thrown, indicating that an undefined identifier had been referenced. As shown in Fig. 5, the AST produced by the syntax analysis stage is *unoptimized*. In addition to decorating `LetExpr` AST nodes with lexical environment information, the semantic analysis compiler stage also performs a pass to optimize the AST.

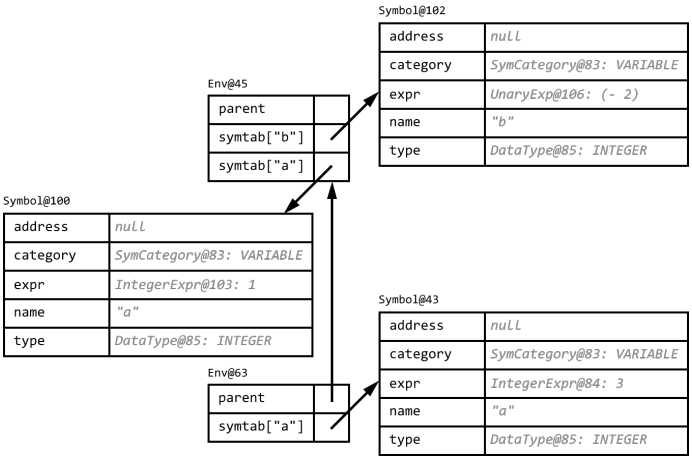

Fig. 7. The lexical environment that is visible to the expressions within the body of the inner `let` expression of the program in Listing 6.

2) The second semantic compiler pass optimizes the AST by reducing complex expressions to more primitive expressions. Fig. 6 shows an example of an unoptimized AST. This AST is unoptimized because it has more nodes than necessary. For example, the `ArithmeticExpr: +` node can be reduced to an `IntegerExpr: 5` node. Likewise, the `ArithmeticExpr: *` node can be reduced to an `IntegerExpr: 20` node. Such reductions are called **constant folds** and are carried out in this second semantic compiler pass.

   Constant folding is an important compiler optimization, for it greatly reduces the number of machine code instructions. Consider again Fig. 6. Rather than completely removing the `ArithmeticExpr: +` node and replacing it with a `IntegerExpr: 5` node, the former node is *decorated* with the latter node. In the implementation of the compiler, the `ArithmeticExpr` class is a subclass of `BinaryExpr`, which implements a `Foldable` interface. AST nodes with the `Foldable` interface have a mutable `fold` property. After this semantic compiler pass, any `Foldable` AST node will have their `fold` property set to an expression. For example, after the compiler pass, the aforementioned `ArithmeticExpr: +` node will have its `fold` property set to `IntegerExpr: 5`. During the development of the Torrey compiler, I have found it much easier to simply add onto the existing AST through decorations rather than rewriting the tree by removing or replacing nodes. In addition to decorating the AST with lexical environments and constant folds, the semantic analysis stage also type-checks expressions and sets the evaluation, or eval, type of `let` expressions, `if` expression, and referenced identifiers.

3) The third, and final, semantic compiler pass performs type checking and determines eval types. In this pass, the compiler checks to make sure the operands to operators are of the correct type. For example, the compiler makes sure that the operands to an arithmetic expression (e.g., `+`, `-`, etc.) are expressions that evaluate to integers. Additionally, the compiler makes sure that the second operand to the `/` arithmetic expression is not zero. If the operands to an operator are not correct, then an error is thrown. Moreover, the compiler type-checks `let` and `if` expressions and realizes the eval types of `let` expressions, `if` expressions, and referenced identifiers. For example, Listing 7 shows the pseudocode of the procedure used to type-check `let` expressions.

```
typecheck(LetExpr expr) {                          1
  prevEnv = top                                    2
  top = expr.environment()                         3
                                                   4
  for (i = 0 to expr.bindings.size - 1) {          5
    typecheck(expr.bindings.get(i))                6
  }                                                7
                                                   8
  for (i = 0 to expr.children.size - 1) {          9
    typecheck(expr.children.get(i))                10
  }                                                11
                                                   12
  top = prevEnv                                    13
  expr.setEvalType(expr.last.evalType)             14
                                                   15
  return expr.evalType                             16
}                                                  17
```

Listing 7. Pseudocode of the procedure used to type-check `let` expressions.

To realize the eval type of a `let` expression, the

compiler first caches the previous lexical environment (in line 2 of Listing 7) before setting the current environment to the `let` expression's environment (in line 3 of Listing 7). With the current environment set, any AST nodes that are visited will now have access to the `let` expression's environment. Next, the compiler visits the expression's *bindings* (in line 6 of Listing 7). A `let` expression **binding** is a mapping of an identifier to an expression. Such bindings are declared within a `let` expression's brackets. For example, the expression `(let [a 2 b 3])` has two pairs of bindings: `a -> 2` and `b -> 3`. When visiting a binding, the compiler type-checks the bounded expression. Once all the expression's bindings have been visited, the compiler visits the body expressions and type-checks those (in line 10 of Listing 7). Finally, the compiler restores the current environment to the cached environment (in line 13 of Listing 7) and sets the eval type of the `let` expression to be the same as the eval type of the last expression in the body (in line 14 of Listing 7). In addition to realizing the eval type of `let` expressions, the compiler also realizes the eval type of `if` expressions.

To realize the eval type of an `if` expression, the compiler first type-checks the expression's test condition expression, consequent expression, and alternative expression (if there is an alternative). Then, if the test condition evaluates to `true`, the compiler sets the eval type of the `if` expression to the eval type of the consequent. Else, if there is an alternative expression, then the eval type of the `if` expression is set to the eval type of the alternative. Else, the eval type of the `if` expression is `UNDEFINED`. Lastly, the compiler realizes the eval type of identifiers.

To realize the eval type of an identifier expression, the compiler searches for that identifier within the lexical scope chain, traversing through the linked list of environments. Once an entry within a symbol table is found, the compiler then sets the eval type of the identifier expression to the data type stored within the symbol. If the compiler has made it past the semantic analysis stage without throwing any semantic errors, then it is safe to assume that the input Torrey program is both syntactically and semantically sound. After this final semantic analysis pass, the compiler takes this sound AST and generates intermediate code.

4) Intermediate Code Generation: **Intermediate code generation** is the final stage of the Torrey compiler's front-end, wherein a pseudo-assembly-like program is derived from an optimized AST. Such a program is written in an intermediate representation, or IR, and is thus called an *IR program*. The syntax of IR programs is shown in Listing 8. IR programs are "pseudo-assembly-like" because they look like assembly programs but have an infinite number of registers (whereas assembly languages have a finite set of registers). An **IR program** is a linear sequence of *three-address code* instructions. A **three-address code**, or TAC, instruction is an instruction that has at most three arguments (typically two operands and an operator). TAC is considered by many to faciliate target-code generation.

```
irprog   -> quad*

quad     -> copy
          | unary
          | binary
          | param
          | call
          | if
          | label
          | goto

copy     -> temp '=' addr
unary    -> temp '=' '-' addr
binary   -> temp '=' addr (arithOp|relOp) addr
param    -> 'param' addr
call     -> 'call' name ',' constant
if       -> 'if' addr relOp addr goto
label    -> 'label' labelid ':'
goto     -> 'goto' labelid

addr     -> temp | name | constant

relOp    -> '==' | '<' | '<=' | '>' | '>='
arithOp  -> '+' | '-' | '*' | '/'
labelid  -> 'l0' | 'l1' | 'l2' | ...
```

Listing 8. The grammar for the intermediate representation, or IR.

The front-end of the Torrey compiler produces TAC because it closely represents lower-level assembly languages and thus facilitates target-code generation. On page 363 in [5], the authors of the Dragon Book assert that TAC is desirable for target-code generation, for multi-operator expressions and nested-control flow are unraveled. For example, `(+ 1 (+ 2 (+ 3 (+ 4 (+ 5 0)))))`, which is a multi-operator expression in Torrey, could be represent by the following TAC:

```
t5 = 5 + 0
t4 = t5 + 4
t3 = t4 + 3
t2 = t3 + 2
t0 = t1 + 1
```

However, due to constant folding, the actual TAC, or IR, is:

```
t1 = 14
t0 = 1 + t1
```

The above could further be reduced to `t0 = 15` by low-level IR optimizations, however, the Torrey compiler has no such optimizations (yet)[8].

[8]Actually, without debugging, I am not sure exactly where such an optimization should be. It could be an optimization that occurs during semantic analysis or code generation. If during the former stage, then I would have to improve the constant folder pass. If during the latter stage, then I would have to add a compiler pass during code generation.

The corresponding x86 is as follows:

```
movq $14, -8(%rbp)
movq $1, -16(%rbp)
movq -8(%rbp), %r10
addq %r10, -16(%rbp)
```

Clearly, TAC, which operates on at most three arguments, greatly facilitates the translation to x86 instructions such as `movq` and `addq`, which operate on at most two arguments. Now that the reader is acquainted with TAC and why it is used in the Torrey compiler, I will continue by explaining how TAC is represented in the compiler's memory.

There are many different ways to represent three-address code in computer memory. In fact, on page 366 in [5], the Dragon Book authors introduce three data structures that can be used to represent TAC: quadruples, triples, and indirect triples; however, I decided to use quadruples. A **quadruple** is a data structure of the form `(op, arg1, arg2, result)`, where `arg1`, `arg2`, and `result` are *virtual addresses* and `op` is an operator. On page 364 in [5], the authors propose three types of such virtual addresses: *name address*, *constant address*, and *temporary address*. A **name address** represents global procedures (e.g., `print` or `println`), a **constant address** represents a constant integer, and a **temporary address** represents a compiler-generated temporary (e.g., `t0`). However, in addition to the three aforementioed address types, the Torrey compiler also uses a *label address* type. Listing 9 shows the implementation of a quadruple as an abstract class. From this abstract class, many types of IR instructions can be created.

```
abstract class Quadruple {
    Operator op;
    Address arg1;
    Address arg2;
    Address result;

    // ...
}
```

Listing 9. A partial pseudocode implementation of the quadruple data structure.

Consider the following Torrey program:

```
(if (< 1 2)
    (print (* 5 5))
    (println (- 32)))
```

From the aformentioned program, the front-end's intermediate code generator produces the following IR program:

```
if 1 >= 2 goto l0
t0 = 25
param t0
call print, 1
goto l1
label l0:
t2 = - 32
param t2
call println, 1
label l1:
```

The corresponding quadruple sequence for the above IR is shown in the table below:

Quadruple subclass	op	arg1	arg2	result
IfInst	>=	1	2	l0
CopyInst	=	25		t0
ParamInst	param	t0		
CallInst	call	print	1	
GotoInst	goto			l1
LabelInst	label	l0		
UnaryInst	-	32		t2
ParamInst	param	t2		
CallInst	call	println	1	
LabelInst	label	l1		

Notice how not all fields are populated; for example, the `CopyInst` subclass of `Quadruple` omits `arg2`. Now that the reader is familiar with the compiler front-end, and the process by which it transforms input strings (Torrey programs) to IR programs (sequences of quadruples), I can briefly explain the default x86 back-end and describe how other back-ends can be integrated.

F. x86-64 Back-end

Recall that a **back-end** is the second part of the Torrey compiler, wherein an IR program, which is produced by the front-end, is translated to a target language. By default, the Torrey compiler targets x86-64 assembly. Thus, the x86 back-end translates an IR program to an x86 program. The x86 back-end performs the following three compiler passes, in order. During the discussion of the compiler passes, consider the following demo program: `(let [a 3 b (- a)])`.

1) The first back-end compiler pass converts the IR program to a pseudo-x86 program. The output of this pass is "pseudo x86" because the memory locations are all compiler-generated temporaries that have yet to be replaced with base-relative stack locations. For example, this first compiler pass translates the IR program of the aforementioned Torrey demo program to the following pseudo-x86 assembly program:

```
.text
  .globl main
main:
  pushq %rbp
  movq %rsp, %rbp
  subq $32, %rsp
  jmp start
start:
  movq $3, t0
  movq t0, t2
  negq t2
  movq t2, t1
  jmp conclusion
conclusion:
  addq $32, %rsp
  popq %rbp
  retq
```

Notice the several compiler temporaries embedded within the program (e.g., `t0`). In the subsequent pass, these will be replaced with base-relative stack addresses. Additionally, notice the labels such as `main`, `start`, and `conclusion`. These labels form a template that helps set up the program's *stack*.

A **stack** is a region of a computer program's memory that gets constantly updated during runtime (that is, the time at which the program runs). Typically, there is one stack with many *frames*, where each frame corresponds to an active procedure within the program. When a procedure starts, a new stack frame is appended, or *pushed*, to the top of the stack. Similarly, when a procedure ends, the frame corresponding to the procedure is removed, or *popped*, off the stack. However, at the moment, Torrey does not support procedures, or functions, and thus there is only one stack frame. In x86-64, there are two special registers that are used to refer to different parts of the stack, `%rbp` and `%rsp`. The former register is the base pointer and points to the bottom of the stack, whereas the latter register is the stack pointer and points to the top of the stack. Now that the reader is familiar with the stack and the two x86 registers used to manipulate it, I can continue by explaining the three parts of the aforementioned template.

The template used by the x86 back-end to provision the stack, which is inspired by what is shown on page 25 of [6], has three parts and is shown in Listing 10. The three parts of the template are: `main`, `start`, and `conclusion`. The `main` part starts on line 3. In lines 4 and 5, the operating system's base pointer is saved to the program's `%rbp` register. Saving the OS's base pointer enables the program to return back to the OS after quitting (on line 14). Next, on line 6, the top of the stack is moved down by n bytes, which effectively creates the stack (n should be a multiple of 16). Then, on line 7, the programs performs an unconditional jump to the `start` section.

```
1  .text
2    .globl main
3  main:
4    pushq %rbp
5    movq %rsp, %rbp
6    subq $n, %rsp
7    jmp start
8  start:
9    # ...
10   jmp conclusion
11 conclusion:
12   addq $n, %rsp
13   popq %rbp
14   retq
```

Listing 10. The x86 program template used by the compiler's x86 back-end.

The `start` section is the location of the main program code. After the execution of the main program code, on line 10, the program performs an unconditional jump to the `conclusion` section. Then, within the conclusion section, the stack pointer is moved up by n bytes, which effectively removes the stack. Finally, on lines 13 and 14, the OS's return address is popped off the stack and then the program jumps to it. Now that the reader is familiar with the x86 back-end's first compiler pass, I continue to describe the subsequent compiler pass.

2) The second x86 back-end compiler pass replaces the compiler-generated temporaries with base-relative stack addresses. This pass simply loops linearly through the list of generated x86 instructions and replaces the temporary addresses with base-relative addresses. After this pass, the previously mentioned pseudo-x86 program is translated to the following x86 program:

```
.text
  .globl main
main:
  pushq %rbp
  movq %rsp, %rbp
  subq $32, %rsp
  jmp start
start:
  movq $3, -8(%rbp)
  movq -8(%rbp), -16(%rbp)
  negq -16(%rbp)
  movq -16(%rbp), -24(%rbp)
  jmp conclusion
conclusion:
  addq $32, %rsp
  popq %rbp
  retq
```

Notice that every compiler-generated temporary has been replaced with a base-relative stack address (e.g., `-16(%rbp)`. The final x86-compiler pass performs an "instruction patch".

3) The third, and final, x86 back-end compiler pass performs an instruction patch. As articulated on page 38 of [6], in x86, each instruction must adhere "...to the restriction that at most one argument of an instruction may be a memory reference." Thus, instructions such as `movq -16(%rbp), -24(%rbp)` are not permitted. Such instructions can be "patched" by simply moving the first argument to a register, say `%r10`, before executing the instruction. Thus, the aforementioned `movq` instruction can be replaced with the following two instructions:

```
movq -16(%rbp), %r10
movq %r10, -24(%rbp)
```

After such instructions have been "patched", the x86

back-end spawns a child gcc process (via the Java ProcessBuilder API) to link the patched x86 program with the runtime object code and to build an executable. The runtime, which is written in C, is rather small and provides standard output support. Finally, the resulting executable can be run on a Linux x86-64 machine. This concludes the high-level technical description of the compiler that implements the Torrey programming language. In the following section, I briefly describe the steps taken to add additional back-ends to target different languages.

VI. Additional Back-ends

As stated numerous times and illustrated in Fig. 4, it is very easy to integrate more back-ends to target additional languages. The following are the steps that must be taken to create an additional back-end for the Torrey compiler:

1) Create a new directory within `torrey/backend/targets/` of the form `arch/vendor/sys`. If the target language does not depend on an architecture, vendor, or operating system (e.g., it is a high-level language), then name these folders unknown (i.e., `unknown/unknown/unknown`).

2) Within the directory created in step (1), create a new class named `ArchVendorSysBackend` that extends the `TorreyBackend` abstract class. As required by the inheritance of `TorreyBackend`, two methods must be implemented: `generate()` and `assemble()`. The `generate()` method generates the target program. Its input is the IR program produced by the compiler's front-end, and its output is a program that implements the `TargetProgram` interface. The `assemble()` method (optionally) assembles the target program into a native executable specific to the target's architecture. This is optional because not every target language needs to be assembled into an executable (e.g., a high-level language). The output of the `generate()` method serves as the input to the `assemble()` method. View the implementation of `X8664PCLinuxBackend` here for an example.

3) Create a new `TargetTriple` to describe the architecture, vendor, and system that is being targeted.

4) Associate the new `ArchVendorSysBackend` with the `TargetTriple` by creating a key-value entry within the registry map in `torrey/backend/targets/Targets.java`. The key is a string representation of the triple and is of the form `<arch>-<vendor>-<sys>`. The value is a reference to the backend that implements the target language for that specific target.

Once the back-end has been created and "registered", it can then be set as the target of compilation via the `--target` compiler flag. To view a list of the supported targets, supply the `--target-list` flag. Now that the reader is familiar with how to add an additional back-end to the compiler, I will continue by describing the incremental approach that was adopted to tackle the complexity that inherently accompanies compiler construction.

VII. The Incremental Approach

When constructing this compiler, I adopted an incremental approach; instead of building just one compiler, I built *three*, each building upon the previous compiler's grammar. Throughout this project, I have read a lot of literature on the subject of compiler construction. I have found that many textbooks adopt a "waterfall" approach to the instruction of compiler construction. That is, the textbook discusses a compiler topic (e.g., code generation), and subsequently moves on to another topic without every circling back. Because of this approach, the chapters tend to be very long, and you cannot get a compiler up and running until you finish reading the textbook. However, during my research on university compiler courses, I came across a textbook that tackled compiler construction from a different angle.

In [6], Jeremy Siek, Professor of Computer Science at Indiana University, offers a unique approach to the problem of compiler construction. In the textbook, the author outlines "an incremental approach" to the construction of a nano-pass compiler for a subset of Racket. In each chapter of the textbook, students implement a new feature in the compiler and make the necessary changes to every compiler stage. Thus, after each chapter, the students have access to a working compiler. I was inspired by Siek's approach and have adapted it for the construction of the Torrey compiler.

In the following sections, I provide the grammars of the subsets of Torrey implemented in each iteration. The last two sections are two proposed iterations that have yet to be implemented.

A. First Iteration: Integers, Unary and Binary Arithmetic, and Standard Output

```
program  -> expr*

expr     -> int
          | unary
          | binary
          | print

int      -> [0-9]+
unary    -> '(' '-' expr ')'
binary   -> '(' ('+'|'-'|'*'|'/') expr expr ')'
print    -> '(' ('print'|'println') { expr }+ ')'
```

B. Second Iteration: Lexically-Scoped Variables

```
program  -> expr*

expr     -> int
          | unary
          | binary
          | id
          | print
          | let

int      -> [0-9]+
id       -> { a-zA-Z_$ }+ { a-zA-Z0-9_$!?- }*
unary    -> '(' '-' expr ')'
binary   -> '(' ('+'|'-'|'*'|'/') expr expr ')'
print    -> '(' ('print'|'println') { expr }+ ')'
let      -> '(' 'let' '[' { id expr }* ']'
              { expr }* ')'
```

C. Third Iteration: Booleans, Logical and Relational Operators, and Control Flow

```
program -> { expr }*

expr    -> prim | id | unary | binary
         | print | let | not
         | and | or | if

prim    -> int | bool
bool    -> 'true' | 'false'
int     -> { 0-9 }+
id      -> { a-zA-Z_$ }+ { a-zA-Z0-9_$!?- }*
unary   -> '(' '-' expr ')'
binary  -> '(' ('+' | '-' | '*' | '/' |
              '==' | '<' | '<=' | '>' | '>=')
              expr expr ')'
print   -> '(' ('print' | 'println')
              { expr }+ ')'
let     -> '(' 'let' '[' { id expr }* ']'
              { expr }* ')'
not     -> '(' 'not' expr ')'
and     -> '(' 'and' expr { expr }+ ')'
or      -> '(' 'or' expr { expr }+ ')'
if      -> '(' 'if' expr expr [ expr ] ')'
```

D. Proposed Fourth Iteration: Globally-Scoped Functions

```
program -> { expr }*

expr    -> prim | id | unary | binary
         | print | let | not
         | and | or | if | fun

prim    -> int | bool
bool    -> 'true' | 'false'
int     -> { 0-9 }+
id      -> { a-zA-Z_$ }+ { a-zA-Z0-9_$!?- }*
unary   -> '(' '-' expr ')'
binary  -> '(' ('+' | '-' | '*' | '/' |
              '==' | '<' | '<=' | '>' | '>=')
              expr expr ')'
print   -> '(' ('print' | 'println')
              { expr }+ ')'
let     -> '(' 'let' '[' { id expr }* ']'
              { expr }* ')'
not     -> '(' 'not' expr ')'
and     -> '(' 'and' expr { expr }+ ')'
or      -> '(' 'or' expr { expr }+ ')'
if      -> '(' 'if' expr expr [ expr ] ')'
fun     -> '(' 'fun' id  '[' { id }* ']'
              { expr }* ')'
```

This (proposed) iteration adds globally-scoped functions to the language, via a `fun` expression. For example, in the following code excerpt, I define a new `square` function via the `fun` expression and subsequently call it:

```
; function definition
(fun square [a] (* a a))
; function usage
(println (square 2))
```

I can also imagine that a preprocessor stage would be prepended to the compiler front-end to enable the import of external functions via an `import` expression:

```
; math.torrey
(fun square [a] (* a a))

; some other file
(import [square] './math')
(println (square 2))
```

E. Proposed Fifth Iteration: Loops

```
program -> { expr }*

expr    -> prim | id | unary | binary
         | print | let | not
         | and | or | if | fun | loop

prim    -> int | bool
bool    -> 'true' | 'false'
int     -> { 0-9 }+
id      -> { a-zA-Z_$ }+ { a-zA-Z0-9_$!?- }*
unary   -> '(' '-' expr ')'
binary  -> '(' ('+' | '-' | '*' | '/' |
              '==' | '<' | '<=' | '>' | '>=')
              expr expr ')'
print   -> '(' ('print' | 'println')
              { expr }+ ')'
let     -> '(' 'let' '[' { id expr }* ']'
              { expr }* ')'
not     -> '(' 'not' expr ')'
and     -> '(' 'and' expr { expr }+ ')'
or      -> '(' 'or' expr { expr }+ ')'
if      -> '(' 'if' expr expr [ expr ] ')'
fun     -> '(' 'fun' id  '[' { id }* ']'
              { expr }* ')'
loop    -> '(' 'loop' '[' id int [ int ] ']'
              expr ')'
```

This (proposed) iteration adds looping via the `loop` expression. This expression is very similar to Clojure's `dotimes` expression, however, the `loop` expression accepts an optional integer to specify the increment. Consider the following examples:

```
; Outputs 0 1 2 3 4
(loop [x 5]
  (print x))

; Outputs 0 2 4 6 8
(loop [x 10 2]
  (print x))
```

Now that I have explained the iterative approach adopted, I will continue by explaining the challenges encountered.

VIII. Challenges

Throughout the development of this compiler, I have encountered many challenges. Building the semantic analysis stage of the compiler front-end was definitely challenging. Additionally, the implementation of the `ErrorReporter`, which is dependency injected into the compiler stage's supporting classes, was quite challenging. Moreover, the parsing of expressions such as `(- 42)` and `(- 4 2)` was quite challenging, for both expressions start with `(-` and are followed by an expression. To sucessfully parse such expressions, the parser first assumes that its parsing a unary expression, and if it fails to parse a second expression, then it's a unary expression. However, if it successfully parses a second expression, then it's a binary expression. Ironically, generating assembly code was the easiest part of the compiler to implement; however, researching the intracacies of the x86-64 assembly languages was very time consuming. Moreover, the implementation of the panic mode errory recovery in the parser is quite hard to get working correctly. In fact, to this day it does not work well. For example, it sometimes produces "ghost" error messages. That is, error messages that do not make any sense and are complaining about something that is correct. The panic mode error recovery enables the parser to continue on parsing without having to stop and throw an error every time it encounters a parsing error. Now that the reader has been informed of the challenges that I experienced during the implementation of the Torrey programming language, I will conclude this paper with some closing remarks and retrospective.

IX. Conclusion and Retrospective

First, I would like to thank my project mentor, Shuhui Yang, for allowing me to pursue compiler construction as a senior design project. Additionally, I would like to thank the Purdue University Northwest undergraduate research committee for approving my research grant. I have learned a lot from implementing a novel programming language, for my understanding of the low-level details that high-level languages abstract away has improved. Without a doubt, running the first executable produced by my compiler was truly the most exciting moment of the entire semester. To continue, I will provide some retrospective remarks.

If I were to start the project over again, instead of using two implementation languages (Java and C), I would have simply used C++. I chose to use Java for two reasons: (1) it is the compiled language of which I am most familiar, and (2) the Java runtime has a built-in garbage collector. However, with the implementation of smart pointers in C++, reason (2) is less convincing. Had I originally implemented the compiler using C++, I would not have had to be dependent on the JVM. Since I implemented the compiler in Java, I am dependent on the JVM, and this is less than ideal when it comes to distributing the compiler in releases. Each release must have the C runtime and the compiler `.jar` file. If the implementation language were just C++, then each release would simply contain an executable for each supported platform.

In the future, I would definitely consider revisiting this project and integrate the lessons learned from this project into the subsequent compiler. Additionally, to improve the ease at which individuals could test my compiler, I would like to build an online "playground". Such a playground would be similar to the Go Playground, wherein users type a program into a textarea and send their code to a server that runs the code in a sandbox environment and sends the standard output back to the user. I can imagine that the playground would be implemented using the client-server model. The client would communicate with the server using either an HTTP REST API or web sockets.

References

[1] Aho, A. V., Lam, M. S., Ullman, J. D., & Sethi, R. (2011). Introduction. In *Compilers: Principles, Techniques, and Tools* (2nd ed., p. 1). Pearson Higher Ed.

[2] Sebesta, R. W. (2002). Evolution of the Major Programming Languages. In *Concepts of Programming Languages* (11th ed., p. 65). Pearson Education.

[3] Rich, E. (2007). Part III: Context-Free Languages and Pushdown Automata. In *Automata, Computability and Complexity: Theory and Applications* (p. 149). Pearson Education.

[4] Appel, A. W. (2002). Translation to Intermediate Code. In *Modern Compiler Implementation in Java* (2nd ed., p. 121). Cambridge University Press.

[5] Aho, A. V., Lam, M. S., Ullman, J. D., & Sethi, R. (2011). Intermediate-Code Generation. In *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson Higher Ed.

[6] Siek, G. J. (2021). Integers and Variables. In *Essentials of Compilation: The Incremental, Nano-Pass Approach.*. Indiana University.