Spring 5-1-2016

The Eagle Programming Language

Samuel G. Horlbeck Olsen
Macalester College, shorlbec@macalester.edu

The Eagle Programming Language

Sam Horlbeck Olsen

Susan Fox, Advisor

Paul Cantrell, Reader

John Cannon, Reader

May, 2016

Macalester College

Department of Mathematics, Statistics and Computer Science

Abstract

C remains the dominant systems programming language despite many new languages attempting to take its place. Modern languages generally value abstraction and safety over speed and direct control of hardware. They are therefore not well suited to the low-level tasks for which C was designed. This paper introduces a novel programming language, Eagle, which represents a fast, elegant alternative to C. It allows low-level programming while providing optional modern features like reference counting, closures, generators, and classes. In addition to specifying this language and reviewing the current alternatives, the paper describes the implementation of a working Eagle compiler. The language and the compiler have reached a state of relative stability and may be considered as a viable option for use in programming tasks, both small and large.

Contents

Abstract iii

1 Introduction 1
- 1.1 Motivation 1

I Programming Language Design and History 3

2 Fundamental Design Theory 5
- 2.1 Programming Paradigms 5
- 2.2 Speed vs Safety 6

3 A Comparison of Eagle to Existing Languages 9
- 3.1 C Influences 9
- 3.2 Modern Languages 10

4 Eagle Specification 15
- 4.1 Types 15
- 4.2 Imports and Exports 16
- 4.3 Control Flow 17
- 4.4 Functions 18
- 4.5 Object Orientation 20
- 4.6 Variables 23
- 4.7 Expressions 24
- 4.8 Reference Counting 24

II Reference Implementation 27

5 Compiler Background 29

5.1 Source Code to Lexical Tokens 30
5.2 Lexical Tokens to Syntax Tree 31
5.3 Syntax Tree to Machine Code 32

6 Tooling **35**
6.1 Lexer and Parser 35
6.2 Code Generation and Optimization 37

7 Parsing and Abstract Syntax Tree **41**
7.1 Imports 41
7.2 First Pass 42
7.3 Parsing and AST Building 43

8 Code Generation **47**
8.1 Utilities 47
8.2 Common Code Generation Process 49
8.3 Step One: Prototype Generation 51
8.4 Step Two: Class Method Compilation 52
8.5 Step Three: Independent Procedure Generation 53

9 Infrastructure **57**
9.1 Handling Multiple Files 57
9.2 Optimization, Assembly & Executable Generation 58
9.3 Memory Management 58
9.4 Containers & Misc. Libraries 59

III Conclusion **61**

10 A Non-Trivial Example **63**

11 What Now? **65**
11.1 What's Working 65
11.2 What's Left 66
11.3 Release Timeline & Long-Term Goals 67

12 Final Thoughts **69**

13 Example Code **71**
13.1 Basic "Hello World" 71
13.2 Object Orientation & Closures 72

13.3 Generators 74

Bibliography **75**

Chapter 1

Introduction

Eagle is a new programming language that aims to take the best aspects of C and present them in a streamlined, easy-to-read syntax, as well as providing modern conveniences to reduce boilerplate code and increase programmer productivity. Eagle was born out of dissatisfaction with the current generation of systems programming languages. It should feel instantly familiar to anyone proficient in C, yet it also has a shallower learning curve for programmers with less low-level experience. It is a *systems programming language* in the truest sense of the phrase: it can be used to write anything from interactive graphical interfaces to operating systems and drivers. It also recognizes the need for modern abstractions and thus provides optional niceties like reference counting and object orientation. This book will review the current generation of modern systems languages, then outline the specification of the Eagle language itself, and finally discuss the implementation details of a working Eagle compiler.

1.1 Motivation

C is the gold standard in low-level systems programming. It can almost be thought of as "portable assembly." But C has been showing its age for quite some time. Over the last several decades, a plethora of new languages have been developed that try to address some of the shortcomings of C. But there are not yet any mainstream languages that carry on the *spirit* of C while also bringing it into the modern era. C gives the programmer as much control over what happens in the machine as possible without writing pure assembly code. Any program requiring that level of control is almost required to be written in C. Most other languages come with complex runtime dependencies or have memory characteristics that make them difficult to justify in time-sensitive applications.

The current trend in programming languages acknowledges the need for improved performance, but tends to miss the mark by attempting to be completely safe. A experienced C programmer understands the pitfalls of unrestricted memory access and allocations, but chooses to continue using the language despite these problems because, in many cases, it is acceptable to trade safety for speed. While safety is a noble goal, it is one factor out of many that determines which language is the best fitted for a task. Eagle is a response to the current trend—it does not guarantee safety, nor does it discourage unsafe constructs. Rather, it leaves these decisions to the programmer. The problem with the current generation of languages is that it does not recognize that there are scenarios in which speed and simplicity are more important than safety. This oversight is one of many reasons why C is still so widely used forty years after it was invented.

If the Eagle project is successful, it will displace C in the niche that C fills. It will cover the same use cases as C and it will provide an almost one-to-one mapping of C constructs onto Eagle code. C programmers will be able to write code in the C mindset, but with a modern, elegant syntax, and without many of the cumbersome elements that come with C's advanced age. Eagle also attempts to expand on C's success; it provides basic memory management via compiler-implemented reference counting, and it introduces modern language niceties and abstractions to the C toolset. Eagle has first-class functions, range-based for loops, generators, and simple classes, all implemented in a way that complements the C mindset and the efficient simplicity that the mindset implies. Libraries written in Eagle should integrate seamlessly into any project written in any language that understands C calling conventions.

If C is the *lingua franca* of programming, Eagle speaks it fluently.

Part I

Programming Language Design and History

Chapter 2

Fundamental Design Theory

For almost as long as digital computers have existed, there have existed programming languages. Programming languages provide for humans a way to communicate effectively with the machine. At a fundamental level, programming languages straddle the line between human logic and digital logic—they bridge the divide between thoughts and bits. Every programming language that expects to be remotely useful must be easy to understand for humans and easy to translate for computers. This chapter will discuss various programming paradigms and will examine some of the trade-offs that are inherent to any computer programming language.

2.1 Programming Paradigms

The major families of programming languages can be organized into several key paradigms that describe the style in which they are used. The most important of these paradigms for the discussion of Eagle are the *procedural* and *object-oriented* paradigms. There are many other types of programming languages that color the tapestry, but for the purposes of this discussion I will focus only on these two (Toal, "Programming Paradigms").

Procedural programming (also known as *imperative* programming) treats blocks of code as the highest-level construct. Every statement that is executed by the machine is contained in a procedure. These procedures can call other procedures and have basic control-flow mechanisms that describe the logic of the particular application. Procedural programming is perhaps the most intuitive of the paradigms as it represents a simple list of instructions read from the top down. Many other styles of programming borrow from the procedural paradigm mainly because it is so intuitive to both humans and computers—it is an easy to follow a recipe.

Object-oriented programming is similar to procedural programming, except that data and functionality are somewhat more unified. The definitions of data types are tied to procedures. Object-oriented programming is very common; it dominates languages like Java and C$^\sharp$. Object-oriented code also stresses *polymorphism*, the idea that multiple types can implement the same interface of functions. There is a lot more to talk about with object-oriented design, but Eagle only has the basic subset of object-orientation that has been discussed here. It should not be considered object-oriented in the same way as Java or even C++. The key for Eagle is the union of data types and functionality.

2.2 Speed vs Safety

Often with programming languages there is a trade-off between speed and safety. This is a very intuitive conclusion. All "programming languages" (in the sense that we use the term) must operate on a digital computer. Such a computer, at a fundamental level, is composed of basic hardware components. This hardware has almost no notion of the higher-level constructs of programming languages that we think of when writing code—the hardware operates via very simple instructions and will do whatever it is told. As such, the raw instructions of the machine are unsafe because they operate without deference to the logical information stored in the machine. Programming languages give some order to these instructions by making sure they are consistent with the information in question, but languages that allow close interaction and manipulation of the machine inherently give up some safety features by allowing unsafe operations to occur.

When the first computers were developed, there were no programming languages in the sense that we use the phrase today. Rather, programmers wrote the raw machine instructions to tell it what to do. This exercise was highly error-prone and thus unsafe. When programming languages began to appear, they ensured a modicum of safety by providing a (leaky) sandbox of types and functions. These old languages (of which C is a member) often remained close to their roots, however, because it was still important to closely associate with the machine architecture.

Safety has become more of a concern, however. Many languages now emphasize safety above all else. But by putting strict requirements on the safety of the language, the designers prevent the low-level machine access that is often necessary to get good performance. In many cases, the performance penalties of that safety do not matter; for certain applications, tight safety controls are critical. But not all applications have such stringent requirements for safety. There have not been many new programming languages that put memory safety in the back seat and say "it is up to the programmer to ensure the correctness of his or her programs." The trend in programming is safety and conservatism, and often in the trade-off between safety and speed, safety wins out. Eagle attempts to address this landscape by introducing a *modern* language that sits closer to the speed side of the spectrum. It is this difference in priorities that makes Eagle compelling compared to many other new programming languages.

Chapter 3

A Comparison of Eagle to Existing Languages

Successful programming languages often leverage the popular and well-understood features of other successful languages in order to be easily understood and adopted. Eagle has many influences from various languages, and also attempts to address many of the same issues as various other languages. This chapter will explore the ways that Eagle is both similar to—and different from—other languages in the field.

3.1 C Influences

At its core, Eagle attempts to mirror the raw power of C. In this sense, Eagle is truly a *systems programming language*—it is possible to compile Eagle files to executable code that does not have any library or runtime dependencies. In this way, it attempts to modernize the syntax of C without removing any of the performance benefits of writing C code.

Eagle is fully binary compatible with C. Eagle functions follow the same calling conventions as C; Eagle can natively call C functions and C can natively call Eagle functions. Furthermore, Eagle can be written in the C style (in other words avoiding all object-orientation and reference counting conveniences) and can therefore be considered as a drop-in replacement for C. In this way, Eagle allows the programmer to have the low-level flexibility that C provides, while *optionally* receiving the benefits of a memory-managed, object-oriented environment. At the fundamental machine level, Eagle can be thought of as a C analog. Unlike other direct descendants of

C, however, Eagle makes a relatively radical departure in syntax and semantics.

C's syntax is a result of years of evolutionary change. A modern language similar to C can take advantage of many of the lessons learned from thirty years of hindsight. Eagle does not require forward declarations and does not have a convoluted type system. It aims to be as clear and crisp as possible; control flow statements do not need parentheses and statements do not need to be punctuated by a semicolon. The primary question that drives the syntax and semantics of Eagle is, "what would C look like if it were designed today with modern compiler technology and massively expanded system resources?"

3.2 Modern Languages

As discussed in the prior chapter, there is a distinct trade-off between language safety and code performance. C and its kin (Fortran, ALGOL, etc.) are extremely efficient at the cost of either safety or usability. At the time these languages were designed, performance considerations were critical since machines had very poor resources. As computers became faster, fledgling languages embraced the idea that speed and power were growing exponentially and thus they trended towards safety and abstraction over efficiency (Fischer). This generation of languages includes Java, Python, Ruby, and many other interpreted languages. C and C++ never lost their market dominance, however because, for many applications, speed and efficiency were still critically important. The most recent generation of new languages aims to be *both* safe and efficient. This section will compare these new languages to Eagle; in the family tree of computer programming languages, these are Eagle's siblings.

3.2.1 Swift

Eagle is perhaps most similar to Apple's Swift language, at least superficially. Eagle's memory management system (compiler-injected reference counting) is heavily influenced by Swift and its quasi-predecessor Objective-C. Like Eagle, Swift compiles to machine code. Swift, however, is driven by fundamentally different goals Eagle. Swift's programming manual states in its introduction,

> Swift is a new programming language for iOS, OS X, watchOS, and tvOS apps that builds on the best of C and Objective-C,

> without the constraints of C compatibility.

Swift likes to think of itself as a systems programming language, but it requires a relatively large runtime library. Thus, Swift is a systems language only the the loosest definition of the term: it compiles to machine code. Swift is fantastic for desktop and mobile development—it is expressive and fast compared to previous generation languages like Java. But it expressly does not attempt to remain compatible with C, and is therefore somewhat difficult to integrate into existing bodies of code written in C. Furthermore, it actively discourages the kind of low-level operations common in C (and required for operating system-level programming).

Due to the difference in design principles, Eagle and Swift generally target different audiences. Yet Eagle does lend itself to higher-level application programming. The key difference is that Eagle treats high-level niceties like closures, objects, and automatic memory management as *opt-in* rather than *opt-out*. In this way, Eagle is self-consistent while exposing an uncluttered C-like interface to the machine. Furthermore, Eagle actively encourages leveraging the raw power of the CPU at the expense of some safety.

Despite these fundamental differences, Eagle and Swift look very much alike. Their feel is quite similar and the stripped down elegance of the C-like syntax of both languages make them a pleasure to code in. Ultimately, barring extreme cases like operating system construction, Eagle and Swift can be used to tackle the same problems and specifications. The difference comes down to developer familiarity and understanding. Someone coming from a Java or Python background might prefer the conveniences and safety afforded by Swift, while a C or systems programmer might prefer Eagle's unfettered access to the machine and its *opt-in* convenience features.

3.2.2 Go

Go is a new "systems" programming language developed by Google. Like Swift, it is only a systems programming language in the sense that it compiles to machine code. Aside from some syntactical points, Eagle and Go are very different. A discussion of Go is included, however, because it serves as design inspiration for many parts of the Eagle language.

Go aims to be both safe and efficient, and holds as one of its chief aims the ability to *easily* write massively parallel programs. Google designed Go with its own server infrastructure in mind. The *Frequently Asked Questions* page of Go's website states,

> We believe it's worth trying again with a new language, a concurrent, garbage-collected language with fast compilation.

Go has a simplified type system compared to other modern languages; it does not have class types and as such there is no concept of inheritance. Rather, it has C-like structures that can (optionally) have methods declared *separately* from them. These structures can implement interfaces, which serve to create a basic form of polymorphism. Eagle is similar to Go in this respect: while Eagle has classes, it does not have sub-typing or inheritance, and polymorphism is done through optional interfaces. Furthermore, Eagle classes are mainly syntactic sugar over structures and functions—Eagle just chooses to make the distinction between classes and structures explicit.

Eagle and Go are totally different in many other aspects. As stated in the language manual, Go is *garbage collected*. This requires that all pointers that point to Go-allocated memory be known to the collector. There is therefore a very strict divide between Go code and any other code executing in the system ("cgo – The Go Programming Language"). The fact that Go does not follow C-style stack layout is a further obstacle to using external libraries from Go code. During external library calls, the stack needs to be copied and reordered in order for any other library not written in Go to work properly. There is relatively significant overhead in this process as demonstrated by Github user Stephen Gutekanst.[1]

Go is a powerful language for certain use cases. In servers, where the latency of a garbage collector is massively overshadowed by network latency, the performance characteristics of a generational garbage collector are perfectly acceptable. For client facing tasks, however, Go is a bit harder to justify, especially when external libraries are used or where time is critical. If a game is making tens of thousands of OpenGL calls (through a C library), there is absolutely no room for the milliseconds of overhead those calls would incur in Go. Thus, Eagle and Go serve entirely different purposes. Eagle aims to be as compatible with external libraries as possible, and transparent to the machine (i.e. no garbage collection), while Go intends to be fast, safe, and easily concurrent. The latter two features of Go almost require that it be run in its own sandbox. While Go and Eagle have similar philosophies regarding types and syntax, they are vastly different in their stated goals and use cases.

[1] Gutekanst developed a program to profile external C calls from Go to determine if batching C calls might make it run faster. The findings are published on Github at https://github.com/slimsag/cgo-batching. The Go documentation does not mention this overhead at all—rather users have discovered it when trying to find bottlenecks in code.

3.2.3 Rust

Rust is a new systems programming language developed by Mozilla. Unlike Go and Swift, Rust can be considered a true systems programming language.[2] The book "The Rust Programming Language" describes the main motivation for Rust in the introduction,

> Rust is a systems programming language focused on three goals: safety, speed, and concurrency. It maintains these goals without having a garbage collector, making it a useful language for a number of use cases other languages arent good at: embedding in other languages, programs with specific space and time requirements, and writing low-level code, like device drivers and operating systems.

More than any of the of the other modern languages discussed, Rust and Eagle overlap in goals and principles. These two languages compete for the same niche. Both Eagle and Rust value "zero-cost abstractions" and low overhead when dealing with other languages and libraries ("The Rust Programming Language").

More than Eagle, however, Rust values the safety of the language generally, and of memory access in particular. Rust has extremely strict ownership requirements for allocated memory. Such a system is in sharp contrast with the C family of languages, which allow unfettered access to memory via raw pointers. Rust does not allow sharing of pointers without following stringent rules. These rules enable the compiler to infer object lifetime in a deterministic way. This makes it nearly impossible (in "safe" Rust code) to dereference a null pointer or spring a memory leak. Eagle attempts to address these problems instead with reference counting and weak pointers. But this is where the two languages diverge most: Eagle does not make any safety guarantees, instead placing the burden on the programmer to *understand* the various lifetime characteristics of objects in memory (i.e. reference counted vs. malloc'd vs. stack).

Eagle and Rust have similar applications, but the style in which the programmer uses them is fundamentally different. Eagle will instantly feel familiar to anyone with a basic understanding of the C family of languages. Rust, on the other hand, has a very steep learning curve as it requires thinking of memory ownership with the semantics of "borrowing" and "lending." The best way to represent the difference between Eagle and Rust

[2] Several projects are already underway to create an operating system written almost exclusively in Rust. *Redox* (http://www.redox-os.org/) is a great example.

is through the direction of constraints: the Rust compiler puts very strict constraints on the programmer—the programmer has no choice but to do things “right.” Eagle, on the other hand, gives the programmer full control and *expects* the programmer to define his or her own constraints in the code.

Chapter 4

Eagle Specification

This chapter will give a basic specification of the Eagle language. This specification is not intended to cover the entire range of the language and syntax; rather it is meant to highlight areas of interest (particularly where Eagle diverges from C). When in doubt, it is best to apply knowledge of C-like languages and refer to this chapter for the more interesting cases.

4.1 Types

Eagle has standard numeric types, but with the distinction that those types are *not* platform-specific. `bytes`, `shorts`, `ints`, and `longs` are 8-bit, 16-bit, 32-bit, and 64-bit, respectively. Unlike C, there is also a built-in `bool` type. There are also the floating point types `float` and `double`, which are 32-bit and 64-bit, respectively.

Eagle, like C, allows types to be decorated with various attributes. The most common is the pointer (`*`) attribute and the counted pointer (`^`) attribute. Unlike C, the attribute is bound to the *type* rather than the *variable*. Counted pointers can be declared "weak" to indicate that they should not bump reference counts. See the *Reference Counting* section below. There are also array types, and these degrade into pointers during function calls. Similar to the other attributes, the array size modifier is placed next to the type rather than the variable.

Procedures have a general type syntax that separates the variable name from the type declaration. Below is a table showing the syntax:

`[inputs : return type]`	C-style function
`(inputs : return type)`	Closure
`(gen    :  yield type)`	Generator

For example, the C function pointer declaration:

```
int *(*function_pointer)(int, double *, int);
```

corresponds to the following Eagle code:

```
(int, double*, int : int*)* function_pointer
```

All of the procedure types must be declared pointers; there is no ambiguity in function pointer declarations. Closures and generators *must* be reference counted, however.

Class types and structure types are referenced by their name (without need of the `struct` or `class` keywords).

4.2 Imports and Exports

Eagle has a standard module import and export system. Code files can declare multiple `export` patterns; global names in the file that match any of those patterns will be visible to other code files that `import` the file. The patterns are wild cards as would be used in a UNIX shell. For example,

```
export 'http_*'
```

will export all names beginning with `http_`. Individual items may be exported by prefixing them with the `export` keyword. For example, a function `sum` could be automatically set to export by having the following definition:

```
export func sum(int a, int b) : int {
    return a + b
}
```

By default, all functions are *not* exported and are therefore local to the file (similar to the effects of the `static` keyword in C). This prevents the cluttering of global namespaces and encourages good encapsulation patterns.

A file can contain multiple import statements, where the text on the line following the `import` keyword is resolved as a code file relative to the file in which the import is found. The code file itself should be imported; there is no need for header files. Circular imports are handled properly. The code from imported files is not joined with the module (in other words it is not a direct copy, as it is in C). Instead, the exported symbols are merely added to the symbol table for the current file. The compiler will still compile the files separately into independent object files. Importing a code file is not sufficient to compile the code contained within it; the compiler must also be executed with that file as input.

4.3 Control Flow

Eagle has simplified control flow structures compared to other C-like languages. The only loop keyword is `for`. Loops can be incremental (like the classic for-loop), conditional (as in a while-loop in other languages), or iterative using the `for .. in` construct. Loops do not require parentheses around the loop statement(s). The following table contains examples of the three types of loops:

Incremental	Conditional	Iterative

`for int i = 0; i < 10; i += 1`
`{`
`    puts i`
`}`
	`for test() == yes`
	`{`
	`    puts ’In␣the␣loop’`
	`}`
		`for int i in range(10)`
		`{`
		`    puts i`
		`}`

The iterative (range-based) loops operate on generators. The syntax and semantics of generators will be discussed in the next section. Note that in the incremental version, each statement of the for-loop is separated by a semicolon. Formally, Eagle requires statements to be terminated by a semicolon. The compiler will inject semicolons at line breaks when it makes sense to do so; unless multiple statements are on a single line, they should not typically be necessary.

If-statements are standard using the `if`, `elif`, and `else` keywords. Parentheses are not required for the conditional clause, and single-line if statements do not need brackets. These statements behave exactly as they do in their C-like counterparts.

Eagle is lexically scoped, so variables declared within a code block (a loop, for example) will not be available outside that block.

4.4 Functions

Functions are the standard unit of execution in Eagle. There are general properties of functions, and then there are some specific types of functions to consider. The structure of a function declaration is as follows:

```
(func or gen) name(parameters) : return type { code }
```

e.g.

```
func main(int argc, byte** argv) : int { return 0; }
```

External function declarations (for example functions contained in a shared library or another object file) follow the same syntax but lack a body and are preceded by the `extern` keyword. In the case of functions returning no value, the return type syntax (`: return type`) is omitted.

Top-level functions are C-style—there is no name mangling and they can be referenced across object files in any language compliant with C functions. The ampersand character (&) can be used in conjunction with a function name in order to get a pointer to that function. In the case of the code below,

```
func main(int argc, byte** argv) : int
{
    var f = &main
    return 0
}
```

the variable `f` has type `[int, byte** : int]*` (see the section about types). Top-level functions are *not* first class, though they can be referenced through function pointers. It is possible, however, to create chunks of code contained in first-class objects—these types of functions are closures in Eagle and will be discussed presently.

4.4.1 Generators

Generators provide a standard means to iterate over a collection. Calling a generator function will return a generator object, which itself can be called repeatedly. In this way they function similar to generators in Python. Below is an example of generators in action:

```
gen range(int max) : int
{
    for int i = 0; i < max; i += 1
    {
        yield i
    }
}

func main()
{
    int i
    var r = range(10)
    r(&i) -- i = 0
    r(&i) -- i = 1
}
```

Calling a generator object directly like this will return a Boolean value, indicating if the procedure in the generator code has completed. As of this writing, generators can only exist as top-level entities; they cannot be embedded in other functions to act as closures.

4.4.2 Closures

Closures use the exact same syntax as functions, but they are nested within other code. Closures have access to local variables internal to the closure function, as well as variables captured from the surrounding scope. Closures require reference counting, and any external variables referenced in the closure are “lifted” to allocated containers that exist for the lifetime of the closure. Because of this lifting, external variables in closures are captured by *reference* and may be modified directly.

Closures introduce the *recur* keyword, which is used for recursive function calls in order to avoid strong reference cycles introduced by referencing the closure variable within itself.

An example of a closure is shown on the next page.

```
func main()
{
    int i = 5
    (:)^ clo = func() {
        puts i
        i += 1
    }
    clo() -- Prints ‘5’; i = 6
}
```

Note the type of variable `clo`. Closures are required to be reference counted, so the `clo` variable is a reference counted pointer to a closure object.

4.5 Object Orientation

Eagle allows for very basic object-oriented code. As the primary design goal of Eagle is “giving C a face-lift,” object-orientation is included merely to allow unification of data and functionality—Eagle by no means requires object orientation, and the simplicity of its object-oriented feature set should act as a deterrent for Java-style object-oriented design. It would be difficult, if not impossible, to write Eagle code in a completely object-oriented fashion. No form of inheritance is supported. Eagle only provides classes and interfaces to implement encapsulation of data and polymorphism respectively.

4.5.1 Classes

Classes are extremely simple in Eagle. They should be thought of as structures with associated functions. Classes are declared in the same way as structures (member variables are declared as structure fields), with the addition of functions. Any function declared within a class declaration (see the previous section) is treated as a *method* of that class. In addition, `init` and `destruct` functions may be declared, omitting the `func` keyword; these functions have special meaning to the compiler but are not required. Inside of methods, an implicit `self` variable is used to reference the current instance of the class.

Opposite is an example of a class declaration:

```
class Greeter
{
    byte* name

    init(byte* name)
    {
        if name
            self.name = name
        else
            self.name = ’World’
    }

    func greet()
    {
        printf(’Hello, %s!’, self.name)
    }
}
```

In compiled code, `Greeter` will be implemented as a structure containing one field: `byte* name`. The compiler will generate a constructor based on the code contained in `init`; the programmer never calls this function directly. The compiler will also generate code for the `greet` method. Whenever the `greet` method is called on a `Greeter` object, the compiler will add an implicit call to the generated function; there is no function pointer lookup so method calls are as fast as native function calls. Class instances are always reference counted.

4.5.2 Interfaces

Interfaces are purely compiler constructions; an interface declaration inside of a source file will not itself generate any code. Interfaces merely define a set of methods that a class must include in order to implement the interface. Interface “objects” can be declared, which allows for basic polymorphism. For example, if `List` were an interface, a variable of type `List^` could be declared; any instance of a class implementing `List` could be assigned to that variable. Interface types are opaque—only functions that the interface defines may be called from interface objects.

Classes declare that they are implementing an interface by using the following syntax:

```
class Arraylist (List, Queue) { ... }
```

In the example above, `Arraylist` would implement interfaces `List` and `Queue`. As in this example, multiple interfaces can be implemented by a single class. The compiler will produce an error if the class does not implement all methods defined by each interface.

Composite interface types are necessary to support this polymorphism. Types may be declared as a list of interfaces separated by the pipe character (`|`). For example, variable

```
List|Queue^ iterable = nil
```

required declares an object which conforms to both `List` and `Queue`; methods from either interface can be called from that variable.

When a variable declared as an interface gets a method call, there is a virtual table lookup that finds the function pointer associated with the correct method. This is necessary as interfaces are class-agnostic. As such, interface method calls are slightly more expensive than pure class method calls and require the Eagle runtime. Virtual tables will be store in static program memory and are generated at compilation time.

4.5.3 Views

Views are somewhat unique to Eagle. They are functions that tell the compiler of possible type conversions. If a view is defined for a class, the compiler will automatically use the function defined by the view to convert objects of the class type to objects of the view type. This conversion is valid for variable assignments, loop iterators, and function parameters. It *will not* work for expressions like arithmetic operations (a view of an `int` will not render the object as an `int` during addition, for example).

Views are best explained with an illustration. Suppose we have a URL class, containing a variety of information; there is a string indicating the base of the URL, as well as any path associated with it; there is an optional IP address field that is resolved for URLs that are web-facing, and several other bits of information. In our program we wish to store all of this information together, hence the need for a class rather than just a simple string. But functionally we would like to treat this URL as a string in certain cases. Opposite is a (stripped-down) example of how this would look:

```
1   class URL {
2       ... -- Other functions, member variables, etc
3       view byte*
4       {
5           byte* buffer = self.viewbuf
6           sprintf(buffer, ’%s/%s’, self.base, self.path)
7           return buffer
8       }
9   }
10
11  extern func http_get(byte*) : byte*
12
13  func main() {
14      var url = new URL(’www.google.com/imghp’, yes)
15      ... -- Misc. setup
16      var webpage = http_get(url)
17      puts webpage
18      ... -- Etc.
19  }
```

At line 11 we declare an external libray function that takes as its sole input a byte array and returns another byte array. At line 16, we pass the URL object to that function. The compiler sees that the function takes a `byte*` and that the `URL` object being passed to the function has a view of a `byte*`. It therefore injects an implicit function call to the view and passes the returned value to `http_get`.

4.6 Variables

Variables are declared as in C:

```
type name [= value]
```

As of the current version, only a single variable can be declared in an individual statement. Uninitialized variables are not zeroed unless they are reference counted (see the *Reference Counting* section). Unless a variable contained in a function is captured by a closure, its type can be treated as an explicit store of a certain amount of memory. In other words, there are no implicit variables under normal conditions in the language. The only exceptions occur when a variable is captured by a closure, when accessing the `self` identifier in a class, and in generators.

Variables that are captured by closures are treated as though they were the declared type, but in the implementation they may actually be pointers to reference counted objects containing the declared type. Likewise with variables in generators, there is a hidden context that contains those variables, so they are also allocated on the heap rather than on the stack. The lifetime of these implicitly counted variables, however, will not last beyond the lifetime of the generator objects or the closures that contain them.

4.7 Expressions

Expressions in Eagle are conceptually the same as most other C-family languages. This section will highlight some of the different types of expressions not present or altered in other languages.

Most of the standard binary operators are available. There is currently no support for bit-operations, though the bitwise operators are planned for future releases. The Boolean operators `&&` and `||` are *guaranteed* to be short-circuited. The modulo operator (`%`) is valid for positive and negative numbers, and will work with floating-point types. There are no increment or decrement operators, but each of the arithmetic operators has an associated assignment operator (i.e. +=).

In Eagle, pointers are dereferenced using the *postfix* ^ operator. In this case, Eagle borrows from Pascal tradition rather than C tradition. Both reference-counted and raw pointers are dereferenced in this way. As such, member access off of a pointer to a struct does not need a special syntax. The arrow operator (`->`) is still present for familiarity, however. Thus, these two statements are equivalent:

```
person^.name = 'Sam'
person->name = 'Sam'
```

All other expressions are nearly identical to their C counterparts, both in form and in function.

4.8 Reference Counting

Eagle provides opt-in automatic memory management via reference counting. This reference counting is provided through a small runtime library that is embedded in every executable. Counted pointers are declared using the ^ symbol, as opposed to raw (C) pointers, which use the standard

* symbol. By default, all counted pointers hold strong references to the pointee. Weak pointers can be declared using the keyword weak before the declaration.

Counted pointers will always be set to nil, even if they are not initialized. Weak pointers will become nil as soon as the object to which they point is freed. These pointers may be unwrapped using the unwrap keyword, but doing so is not recommended—there are absolutely no guarantees about lifetime, and unwrapping will not increment the reference count.

Any type can be reference counted, including numbers, structures, other pointers, and classes. There is a general understanding between programmer and compiler that reference-counting will work (and prevent memory leaks and double frees) as long as they are not explicitly unwrapped—any code that follows these guidelines but exhibits memory issues should be treated as a bug in the compiler. Due to the strict rules, counted pointers and raw pointers exist in different universes; if necessary, multiple functions need to be declared that accept both counted and raw pointer values—the generated code inside of these functions will be significantly different.

The memory referenced by counted pointers cannot be arbitrarily created; rather the new keyword is provided to allocate the memory and start the reference counting. The syntax for creating counted memory is as follows:

```
 new type[(initializer)]
```

e.g.

```
 int^ i = new int
 int^ j = new int(42)
```

In the case of variable i, the pointer i is initialized to an allocated chunk of memory storing space for an int; the allocated memory contains random data (as in a direct call to malloc). In the case of j, that memory is also initialized to the value of 42. The initializer contains only a single value for primitive types, and may contain multiple values (separated by commas) for classes. Classes are a special case: parentheses are always required—the values are passed to the init function. For classes, using new should be thought of as calling a function.

The runtime that handles reference counting can accept *valid* counted pointers and nil values. Setting a counted pointer to any other type of value (or to a raw pointer) results in undefined behavior (most likely as a segmentation fault).

Reference counting is powerful and is, in general, safer than keeping track of memory manually; it is only in the edge cases outlined above that the programmer may run into problems. The use of reference counting is actively encouraged when programming in pure Eagle code—if code is not going to interact with external libraries often, there should hardly ever be need to call `malloc`.

Reference counts can be manipulated directly using the `__inc` and `__dec` keywords. Using these keywords is nearly as dangerous as using the `unwrap` keyword. For every place a pointer is `__inc`'d, somewhere there needs to be an matching `__dec`. These tools are useful mainly for data structures where reference pointers pointers need to be stored in non-reference counted structures. It would be bad form to declare a public interface that requires other programmers to use `__inc` and `__dec`.

Part II

Reference Implementation

Chapter 5

Compiler Background

As part of the language specification, I am developing a *reference* compiler that translates Eagle code into machine code using the Low Level Virtual Machine framework (Lattner). When I refer to a *reference compiler*, I make a conscious distinction between the language and the program that translates code written in the language to machine code. The former is mostly theoretical work while the latter is a more straightforward programming task. Many compilers may exist for any single language. However, in the absence of any formal specification (aside from that in the early section of this book), the behavior of the compiler I discuss in these subsequent chapters should be treated as the *de facto* standard for the language. This chapter will discuss some of the common procedures and practices common to compiler design.

Compilers are complex tools with many moving parts. Fortunately these systems lend themselves well to modular design. Almost every compiler follows the same basic process. Input text is "lexed" (examined by a lexical analyzer and turned into streams of tokens), then parsed, and finally translated into executable machine code understood by the CPU. The goal of this flow is to turn information-dense raw text code (the language, in this case Eagle) into something more easily modified and traversed in code—usually some sort of syntax tree—and finally into something that the computer can execute (Mogensen, 2–3). Below is a schematic of this process.

All of the blocks in Figure 5.1 represent forms of data that contain equivalent information. In other words, no information is lost or gained between these blocks; rather they are translated from one form to the next. This process is, in essence, the primary concern of the compiler. Each arrow

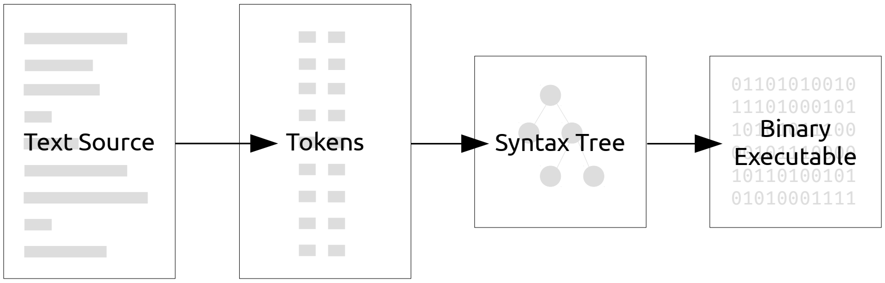
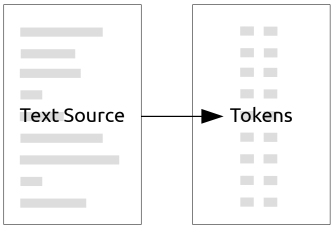


Figure 5.1: The stages of data evolution in a compiler

represents a distinct compilation phase that is logically separate from the others. This distinction is possible precisely because each from is equivalent in information; the source can be wholly transformed into tokens after which point the source is no longer needed. The importance of each of these stages will be enumerated in the following sections.

5.1 Source Code to Lexical Tokens



The process of converting source code into an executable program begins with a process known as "lexing." Lexing refers to lexical analysis of source code and the conversion of text into tokens. Tokens are simply numeric tags (usually enumerated with names in the compiler source) representing various kinds of text.

For example, Eagle has a *reserved* word whose spelling is `sizeof`. Inside the text source, the text "sizeof" has no special meaning. During the process of lexical analysis, the lexer would read from the input file (here the box on the left) a string spelled "sizeof" and would output an enumerated token which has a value inside the compiler of `TSIZEOF`. In this case, "sizeof" is a concrete spelling of a word. The lexer can recognize entire classes of text as well (Mogensen, 10–11). For example, an identifier is anything that is not reserved and that contains letters and potentially numbers. Anything that follows that rule is considered an identifier. Thus, through a system of rules and definitions, the lexer converts raw text into a stream of tokens:

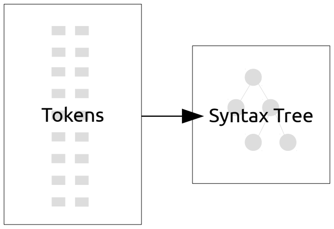
```
    int pointerSize = sizeof(byte*)
```

Becomes,

```
    TINT TIDENTIFIER TEQUALS TSIZEOF TLPAREN TBYTE TSTAR TRPAREN
```

In this way, the rest of the compiler need not deal with the difficulties of string manipulation (which is especially painful with the C implementation of strings), and can instead operate on streams of numbers. This stream is fed directly into the parser, which handles the next step of translation.

5.2 Lexical Tokens to Syntax Tree



The stream of tokens generated by the lexer is still relatively close to the source input. The compiler would have a very difficult time generating machine code if it had to work directly with this stream of tokens. Instead, it relies on the "parser" to generate a more meaningful data representation.

Syntax trees have a very loose definition. They are typically called "abstract syntax trees" because they can be built in many different ways and can have very diverse structure. As a general rule, however, they often have nested levels of trees with each tree and subtree representing some sort of language construct (Mogensen, 99). For example, a binary operation would have two sub trees and an operator. Those subtrees could be any other expression (including another binary tree). This tree could itself be contained in another tree—a loop tree, for example. This creates a hierarchical data structure that expresses the full meaning of the source code. It also allows the compiler to represent operator precedence in a non-ambiguous way. The example from above can thus be expanded to the tree shown in Figure 5.2

The entirety of the source code can be represented in this tree. Every type of node has different entries depending on what construct it represents. Every function is represented at the top level by a single node. Tree nodes also act as linked list nodes; instead of having a child node for every single operation inside a given function, the function node has a single child representing the function body and every subsequent tree inside the function is linked in a list structure from that body child node.

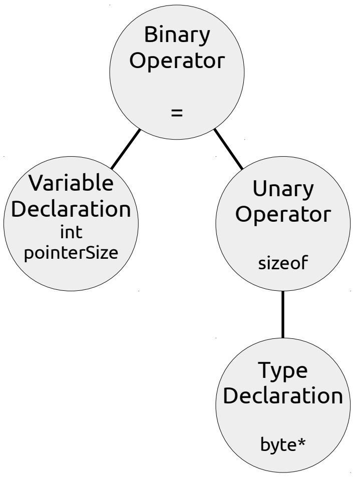
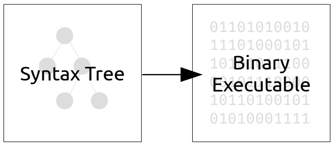


Figure 5.2: Syntax tree representing: `int pointerSize = sizeof(byte*)`

Every compiler will have a different syntax tree format; it is purely an intermediate representation that is (usually) independent of both the language and the lexer (Mogensen, 100). So far in this flow the text has been turned into a stream of tokens (easily understood by the parser) and the parser has turned those tokens into a hierarchical tree structure that is easily walked by the rest of the compiler. The compiler will continue from this point onward without needing to look at either the source form or the token form again.

5.3 Syntax Tree to Machine Code



Once the syntax tree has been created, the actual process of generating something understandable by the CPU can begin. The tree represents the program in a way that removes any ambiguity about precedence, so a simple walk of the tree will generate code with operations in the correct order.

At this stage, some compilers differ. It is relatively common to convert the syntax tree into an *intermediate representation* (IR). IR code is often some form of simplified assembly or a list of basic instructions. Other compilers simply walk the tree and generate code immediately (Mogensen, 147). As will be discussed in subsequent chapters, the Eagle compiler uses libraries to generate actual machine code; it therefore generates IR code that is understood by that library. In general, however, this stage can be thought of as a self contained syntax tree to machine code step. Optimization often occurs at this stage as do various other forms of analysis. All compiler errors other than syntax errors are generated during the walking of the tree.

5.3.1 Optimization

A very common (although technically unnecessary) aspect of the code generation stage is optimization. Humans are good at writing code that other humans understand. Computers work differently than our brains, so it is often lucrative to analyze the syntax tree or IR code to find places where the machine can simplify things. A loop, for example, can be *unrolled* to maximize CPU performance. Whereas to the programmer it makes sense to express a loop as a series of discreet iterations, the computer just sees the loop as a series of instructions followed by a jump to an earlier part of the code. Jumps can slow performance, so to make the code run faster, the compiler may unroll the loop by putting several iterations of the loop inside of each jump. The executable gets bigger but there may be a bit of a speedup. This is the essence of optimization.

There is an informal standard for optimization in mainstream compilers. Compilers will often give the programmer options for optimization level. There may be dozens of classes of optimization, each with varying degrees of aggressiveness. Rather than having the programmer be responsible for specifying each of these optimizations manually, compilers will often bundle them together in *optimization levels*. These levels are expressed in the compiler command line switches `-O0, -O1, -O2, -O3` or `-Os`. Each step represents an increasingly aggressive optimization level. This "standard" is implemented by many mainstream compilers like GCC and Clang.

There are many types of optimizations and studying optimizations could be an entire honors thesis unto itself. A deep understanding of the process of optimization, however, is not necessary for the rest of this paper. As I will describe in subsequent chapters, the Eagle compiler uses external libraries that handle both code generation and optimization. As such, this stage of the compilation process runs almost entirely in a black box—the Eagle compiler knows about optimization levels but it does not concern itself with their implementation.

Chapter 6

Tooling

The Eagle reference compiler uses several tools to aid in the code generation process. The Eagle compiler is written entirely in C with a dash of C++. The usage of C guarantees extremely fast code and modular design, and allows the compiler to be built on a wide range of architectures and operating systems. I considered rebuilding the compiler in Eagle code, but this change was ultimately rejected as the process of *bootstrapping* the compiler (rewriting the compiler in the language it targets, in this case Eagle) would have been too arduous when porting to new CPUs. In the relatively short amount of time I had to develop the compiler, the marginal returns from writing the compiler in Eagle were not enough to justify making such a switch—instead I was able to spend more time focusing on adding critical language features. The biggest reason for using C, however, is the fact that it integrates nicely with all of the tools and frameworks (including the lexer, parser, and code generator) used to build the compiler.

The Eagle compiler uses *GNU Flex* (Lesk and Schmidt) for the tokenization and lexical analysis; for parsing, the compiler uses *GNU Bison* (a variant of *YACC*, Yet Another Compiler Compiler; Johnson). For machine code generation, the Eagle compiler uses the Low Level Virtual Machine (*LLVM*). The details of these tools will be discussed in the following sections. Actual discussion of how the compiler uses these tools can be found in the next chapter.

6.1 Lexer and Parser

Flex and *Bison* are relatively old technologies, but they have proven useful in creating the basic compiler for this project. Both are based on technologies created for the original UNIX systems to aid in the design of compilers. I chose to use these tools for the Eagle compiler because they have a proven track record and they are extremely common among UNIX-derived systems. Furthermore, both are easy to use and go a long way in simplifying compiler construction. It is fairly common to hand-design parsers for programming languages, but in my case the chief aim was to get a robust, working compiler finished in a short amount of time.

6.1.1 Lexer

Flex is the GNU variant of the lexical analyzer generator *Lex* first described by Lesk and Schmidt. *Lex* was proprietary code and was thus re-engineered by programmers with the GNU Project; in practice the two programs work almost identically.

Flex takes as its input a file containing a domain-specific language and generates a C code file whose purpose is to tokenize and analyze text. The details of the language used to generate the lexical analyzer can be found in Lesk and Schmidt. At a fundamental level, Flex uses regular expressions to match tokens on an input stream. The API user (in this case the Eagle compiler) specifies the regular expressions and bits of C code to be executed when the expressions are matched. In practice the code mainly decides whether or not to save the contents of the text (important for passing on the value of a string literal, for example), and assigns an enumerated token value that is ultimately consumed by the parser.

For the Eagle compiler, Flex is used to recognize classes of common tokens, as well as keywords. For example, Flex allows the compiler to distinguish between integer types, decimal types, and identifier types in a generic way. Integers are just strings of digits, decimals are digits with a period somewhere in the text, and identifiers begin with a letter and are followed by a string of either letters or digits. Nothing much more special happens in the lexer; it mainly consists of a list of keywords and associated tokens. Recall Figure 5.1 and the discussion in chapter 5.

Flex works extremely well for the purposes of the Eagle compiler. It allows the compiler to get up and running relatively rapidly. The biggest drawback is that there is inherently less control using this tool than if I were to write the lexer myself. There is also a lot of global state used by the lexer which precludes the possibility of multicore compiling. Ultimately, however, I found these drawbacks were not enough to warrant a full rewrite using completely hand-made tools.

The lexical analyzer is not typically used as a standalone component;

rather it works closely with the parser. In the case of the Eagle compiler, the parser is also an historical UNIX tool, *yacc*.

6.1.2 Parser

Bison is the GNU variant of the *yacc* parser generator. Yacc was introduced as a tool to aid the design of compilers by creating a simple way for programmers to specify a *context-free grammar*[1] and get a functional parser as a result.

Bison generates an LALR(1) parser. Such a parser is relatively old technology but can still describe complicated grammars. The implementation details of the parser are beyond the scope of this paper, but at a fundamental level LALR(1) parsers use a lookup table that decides to *shift* or *reduce* the sentence being built based on the input stream. LALR(1) parsers are quite fast as a general rule, but due to their nature care is needed when defining grammars—the *shift/reduce* decision introduces some ambiguities (Mogensen, 88).

The grammar that is specified as part of the Eagle compiler uses enumerated tokens that are delivered as a stream as part of Flex when building sentences.[2] Flex and Bison work closely together and may be seen as one integrated unit. As part of the compilation process, C files for both the lexer and the parser are generated.

Flex and Bison are very powerful tools that allow for the creation of robust, professional compilers in a short amount of time. They are not ideal for many tasks, but ultimately they only really fail in the edge cases. In terms of the Eagle compiler, these tools are invaluable—they are well-documented, stable, fast, and much easier to use than an inherently-buggy hand-made parser and lexer.

6.2 Code Generation and Optimization

The Eagle compiler uses the Low Level Virtual Machine library (LLVM) for machine code generation. LLVM was first described by Lattner as a machine-independent instruction set that acts as an intermediate step between high-level languages and the hardware. The goal of the library is to

[1] A context free grammar is another theoretical construction used to describe formal language grammars. I will not go into a lot of depth describing how these grammars work. The most important point is that they are *context-free*, meaning they can only describe grammars where the surrounding lexical context does not matter.

[2] See `parser.y`

assist in the creation of compilers by providing a modular, universal backend. It is maintained by Apple and is used as a backend for their Clang compiler (now the default C compiler on all Apple machines). It is also used as the backend for the main compilers of both the Rust and Swift languages described in Chapter 3. What Flex and Bison attempt to do on the front end, LLVM does on the back end. There are numerous benefits associated with the LLVM framework, and this section will attempt to enumerate them.

6.2.1 API

LLVM is provided as a set of modular libraries built as a collection of shared object files. Though there is an overarching superstructure, only a small part of the framework need be loaded at any given time, depending on the use case. For example, the Eagle reference compiler uses perhaps ten percent of the available LLVM libraries.

The LLVM core is written in C++, and the cutting edge API is provided as a C++ library. In the early stages of the development of the Eagle compiler, it became abundantly clear that the C++ library was too unstable across LLVM versions to be relied upon by a single developer. The LLVM project also provides a C API for compatibility purposes (C is often the lowest common denominator, especially on embedded systems), which aims to be much more stable across LLVM versions ("LLVM Developer Policy", *C API Changes*). LLVM is currently developed by Apple and is closely tied to their various projects like Swift and Objective-C (via Clang). While those compilers are developed in tandem with the LLVM framework, the Eagle reference compiler is on a much longer development cycle. Therefore the stability of the C API is very attractive.

6.2.2 Code Generation

LLVM provides a machine-independent assembly-like language. As such, new compiler front-ends need only target the LLVM "machine," thereby targeting all CPUs supported by LLVM. LLVM currently supports x86, x86-64, ARM, SPARC, Power-PC, and many more machine architectures ("Getting Started" (LLVM), *Hardware*). By using LLVM, the Eagle compiler can produce executable code that will run on any of these architectures.

To assist in the creation of LLVM assembly code, LLVM provides an instruction builder API. This API provides a simple way to create the LLVM assembly by abstracting the string manipulations and ordering problems inherent to this type of problem. There are API calls that generate all types of LLVM assembly instructions, and each instruction has an inherent value. Thus, instructions can reference other instructions by using the values returned during creation. Below is an example of this concept in pseudocode:

```
a := LLVMBuildAdd(3, 4)
b := LLVMBuildAdd(a, 5)
```

In this example, LLVM would create machine code representing first `3 + 4` and then the result of that addition and `5`. `a` and `b` both represent instructions, but they can be treated as values for other instructions.

The instruction builder API also provides some helper utilities to make code generation easier. It specifies a uniform way of defining record types like C `structs`. It also allows for the manipulation of instructions, the alteration and interchange of code blocks, and search functionality. The usefulness of these facilities will be exemplified in the next chapter during the discussion about closure variable lifting. LLVM manages this intermediate form internally—the compiler writer can think of this process as translating one syntax tree to another.

6.2.3 Optimization / Executable Creation

By default, LLVM does not support the creation of machine code directly—while the instructions are constant across CPU architectures, the format of executable files differs among operating systems. Therefore the system assembler and system linker must be used. There is experimental support for executable file generation within LLVM itself, but it is still buggy and has many pitfalls. Assembling code and linking it is therefore separate from LLVM and will be discussed in Chapter 9.

LLVM has a large library of potential optimizations that can be run on each source file. While the compiler writer can pick and choose these optimizations, it is usually better to simply provide options to compile with the *de facto*-standard optimization levels (`-O0, -O1 ... -Os`). The Eagle reference compiler uses these optimization flags. The optimizations are applied to a code module (the output from the instruction builder), and at this point assembly code can be produced. As I discussed earlier in this section, LLVM does not actually produce executable code that the machine can run. Assembly code is as close as it gets. For the purposes of the compiler, this is more than enough—the most difficult task takes place during the generation of the assembly code, not during the assembling and linking phases. Thus, at this point in the compilation process, LLVM ceases to be useful. The details of the *shipping* part of the code generation process (assembling and linking) will be explored in Chapter 9.

The LLVM project provides an indispensable part of the compiler process, allowing individual compiler writers to skip the arduous process of assembly code generation and freeing them to focus more on the semantics of the new language.

Chapter 7

Parsing and Abstract Syntax Tree

As discussed in chapter 5, the first stage in compiling source code is converting the string data of a code file to something more understandable for the computer—in many cases an *abstract syntax tree*, or AST. The Eagle compiler uses *GNU Bison/Flex* to generate the parser/lexer. The code for those tools may be found in `src/grammar/eagle.{y|l}`. This chapter will not discuss the details of the grammar; rather it will focus on the steps necessary to generate a workable AST for the rest of the compiler to translate.

When discussing the fundamentals of compiler design, I described the flow of information as relatively one-directional and modular. In the actual implementation, idealism gives way to pragmatism and various parts of the compiler need to talk back and forth. Since Eagle allows other code files to be imported, the compiler needs to fetch a list of the valid symbols (names) contained in those imported files. At this point, the compiler also needs to run multiple passes in order to recognize tokens as either names or identifiers. The details of importing code and running multiple passes is explained in the rest of this chapter.

7.1 Imports

An important distinction between Eagle and C is the role of header files. In C, the programmer must provide separate, distinct header files with function prototypes in order to share code across files. Eagle also requires forward declaration of external functions, but it removes the burden of manual header creation from the programmer. Instead, the compiler autogenerates header files and includes them behind the scenes. This process uses only the lexer to glean the necessary information needed to created the header. Ultimately the compiler only needs to know name and type information from the included source file. The entire parsing mechanism is therefore not needed. By simply running the lexer over the file, tokens like `TFUNC` can be saved along with the rest of the function definition (which follows a well-defined pattern), and the requisite information can be stored without invoking the parser.

A side effect of the way I have implemented the importing library is that types cannot be fully understood at this stage (recognizing pointers, for instance, would require the functionality of the full parser). As such, the header generation can only copy symbols; it is not very intelligent. Rather, it sticks "extern" token in front of every exported symbol and lets the parser handle the actual meaning during the final pass.[1]

Header code is stored in allocated strings on the heap. A special *multibuffer* module allows the lexer and parser to draw from an arbitrary number of files and strings (see Chapter 9). Thus, if a single file is imported multiple times in a single invocation of the compiler, the generated header can be kept around in memory and chained using the *multibuffer*. This method also prevents cycles in imported code.

7.2 First Pass

Eagle's grammar is not purely context-free; the compiler must be able to differentiate between types and identifiers. A class name, for example, is a type (and can be arbitrarily named, like an identifier), whereas a variable name is truly an identifier. In order to properly parse the syntax, type names and identifiers need to be semantically different. The following snippet of code represents a rule in the parsed grammar and demonstrates the need for this differentiation:

`variable declaration` is: `TTYPE TIDENTIFIER`

Such a syntax would need to recognize:

```
double count
```

where `double` is a `TTYPE` and `count` is a `TIDENTIFIER`. But suppose we had a class `ArrayList`—`ArrayList` would need to be a `TTYPE` in order for the grammar to make sense. ArrayList is not a built-in type, so the lexer needs to somehow determine if the word "ArrayList" is an identifier or a type.

[1] See `src/environment/imports.c` for the full implementation.

The solution is to run a first pass over the code to collect type names. Struct names, class names, and interface names are found by using the lexer to extract each individual token. There is no need to invoke the parser itself in this process. Once a type name is found, it is added to a *symbol table* representing named types. During the process of AST building, the lexer uses this lookup table to determine if a string of letters is an identifier or a type. Figure 7.1 demonstrates this decision process. At this point the compiler does not care about the actual type of these names; it is solely concerned with distinguishing type names from identifiers.

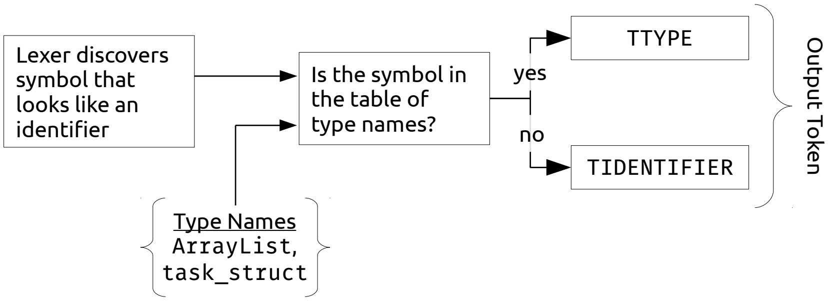

Lexer discovers symbol that looks like an identifier
Is the symbol in the table of type names?
yes
no
TTYPE
TIDENTIFIER
Output Token
Type Names
ArrayList,
task_struct

Figure 7.1: The decision tree for recognizing identifiers versus types

Once the first pass is complete, the code can be properly parsed into an AST.

7.3 Parsing and AST Building

As discussed in the Tooling chapter, the Eagle compiler uses Bison to generate the parser. Programs are divided into declarations; declarations are divided into statements; statements are divided into expressions. Figure 7.2 shows a small subset of the parser definition to give an idea of what this code looks like.

The rules in the parser definition are a combination of tokens (as per the discussion about the lexer) and of other rules. At a basic level, all rules can be broken down into basic tokens. But with these rules the compiler can build up complex structure to describe a valid program. The actual implementation is much more filled out than the basic example in Figure 7.2—rather this should serve as an example for thinking about how the parser works to create syntax trees. Each line in the parser definition (and

Rule	:=	Definition
type	:=	TINT
	|	TDOUBLE
	|	TBYTE
	|	TTYPE[a]
	|	type TSTAR[b]
variable_decl	:=	type TIDENTIFIER
decl_list	:=	variable_decl
	|	variable_decl TCOMMA decl_list
func_ident	:=	TFUNC TIDENTIFIER TLPAREN decl_list TRPAREN

Figure 7.2: A simple subset of the parser rules

[a] An "identifier" that is registered in the type table
[b] A recursively-defined pointer type

thus each line in Figure 7.2) can specify C code that will execute when the line is matched. In this way, every line comes to represent the creation or modification of a syntax tree node.

The abstract syntax tree is represented as a series of C structures with a common, generic header. Every major expression type has its own tree node representation. The tree representation of the code is strung together in the parser. The definitions of the syntax tree nodes are all contained in a single file and are relatively self explanatory.[2] There are tree nodes for the main control-flow constructs (loops, if/else blocks, functions and returns), there are tree nodes for expressions (binary and unary operators, and function calls for example), and there are tree nodes for top-level constructs like classes and structure declarations. There are also miscellaneous other tree nodes that are used for various rules in the grammar.

The syntax tree generation stage has a uniform API that I designed—the code in the parser definition makes calls to functions beginning with `ast_`. The parser generator has various means to allow the building of complex structures easily. It allows parts of rules to be referenced using `$#` syntax. For example, the rule

```
    expr := expr TPLUS expr { $$ = ast_make_binary($1, $3, ’+’); }
```

would tell the parser to call the function `ast_make_binary()` with the expression on the left and the expression on the right of the plus sign, specifying the binary operator to be plus. Assigning to `$$` assigns the output of the rule for the parser—subsequent rules can use `expr` as a tree node as that is the returned type from the `ast_make_binary()` function. With all of these pieces in place it is extremely simple to build up the syntax tree. Once this process is done actual code generation

[2] See `src/compiler/ast.h`

may begin.

Chapter 8

Code Generation

At a high level, code generation is a relatively simple process. The compiler recursively walks the syntax tree, dispatching various compilation functions depending on the type of node encountered. All of these functions return LLVM values, representing instructions in the IR code. By stepping through every node in the tree and recursively walking down them, the compiler builds up the IR code naturally through calling these functions. At the end of this step a full IR module has been built. This process is broken up into several steps.

This chapter will discuss the important aspects of those steps and will highlight some of the more difficult and convoluted code generation aspects. I will start by discussing some of the utilities inside the Eagle compiler (aspects that are related but not directly tied to the compilation process), before talking about the specifics of code generation. The compilation stage can be broken into several discreet steps, which I will enumerate in this chapter.

8.1 Utilities

There are several important utilities that exist independent of the code generation stage, but which are crucial to that process. The way that the Eagle compiler manages types and variables is necessary for the discussion of code generation.

8.1.1 Types

The main data types that Eagle understands were highlighted in Part I. In order to represent those types, the Eagle compiler uses a set of `structs` with uniform headers. These structures are operated on by a series of helper functions that provide a variety of actions. LLVM has a relatively rigid type system, so the types that the compiler uses mirror the LLVM types wherever possible. At any point during compilation, an Eagle compiler type may be directly converted to an equivalent

LLVM type. Struct, interface, class, and function names are kept in globally visible containers, mirroring their global visibility in the input code.[1]

The fundamental unit of the type system (and the header that every type subclass contains) is given by the following struct definition:

```c
typedef struct {
    EagleBasicType type
} EagleComplexType;
```

where `EagleBasicType` is an enumerated value representing an identifier for the more complex type defined. There are many fundamental (built-in) types for which the simple `EagleComplexType` structure is sufficient. These include the range of integer types

(`ETInt1`, `ETInt8`, `ETInt16`, `ETInt32`, and `ETInt64`), as well as floating point types. There are also more complex types that have their own information associated with them. Pointers (enumerated `EagleBasicType ETPointer`), for example are represented through a subclass of `EagleComplexType`:

```c
typedef struct {
    // Shared header with EagleComplexType
    EagleComplexType type;

    // Type-specific information
    // (in this case for the pointer type)
    EagleComplexType *to; // Pointee type (e.g. ‘byte’ in ‘byte*’)
    int counted;       // Is it reference counted?
    int weak;          // Is it declared weak?
    int closed;        // Does it reference a captured variable?
} EaglePointerType;
```

Eagle has a relatively rigid type system, so pointer types need to know the specific pointee type. Pointers types also may or may not be in the reference counted regime, and that difference needs to be distinguished in the type system. Similar type-specific elements are included for the other complex types (function types, class/structure types, interface types, etc.). Wherever possible, types are reused to save memory allocations.

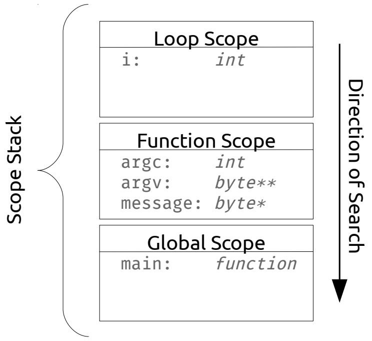
[1] The full type implementation may be seen in `src/core/types.c`.

8.1.2 Variable Management

Eagle has strict scoping semantics, much like C. Variable names are stored—along with associated types—in banks of tables. Whenever a new scope is encountered (for example under a loop), a new table is pushed onto a stack. This stack is walked from top to bottom whenever a variable name is encountered. A nice property of this system is that it can be managed independently. For example, it is possible to execute callbacks when an outside variable is referenced within a closure scope. The intricacies of variable management will be discussed in the next chapter. Figure 8.1 shows a side-by-side example of scope stacks for a basic loop.

```
1 func main(int argc, byte** argv)
2 {
3     var message = 'Hello, world!'
4     for int i = 0; i < argc; i += 1
5     {
6         puts message
7     }
8 }
```

(a) Eagle code



(b) Scope stack

Figure 8.1: An example of variable resolution (compiling at Line 6)

It is important to note that, while variable scope is managed through a stack-like structure, during compilation all memory allocation for variables is done at the beginning of the containing function. This allows a certain class of optimization to occur, but has no effect on the semantics of the language.

8.2 Common Code Generation Process

In general, the code generation within the Eagle compiler is relatively straightforward. The abstract syntax tree is translated into something that LLVM understands. This section will first take a brief look at the basic steps for LLVM code generation. All subsequent steps of compilation use the principles outlined in this section.

LLVM has API calls for every instruction allowed per the specification. It also has a uniform system for declaring constants. All instructions are treated as LLVM values. In this way, variables and instructions can be referenced using a handle to a value reference. Code is broken into "basic blocks," which can be referenced in, for example, jump instructions. A function is simply a large chain of these value-based blocks, each of which is subsequently composed of a large chain of value-based instructions. All type information is contained in the compiler and the LLVM context.

Every instruction has its own API call. Basic blocks also have a special API. The resulting code is quite straightforward. Below is an example of C code which generates a function taking two arguments, returning their sum.

```
LLVMValueRef func = LLVMAddFunction("sum");
LLVMValueRef block = LLVMAppendBasicBlock(func);
LLVMValueRef sum = LLVMBuildAdd(LLVMGetParam(func, 0), LLVMGetParam(func, 1));
LLVMBuildRet(sum);
```

This code is fairly boilerplate and most of the compiler consists of it. The rest of this section will focus on the more interesting cases. All basic instructions in the LLVM IR code are generated using functions similar to these. There is *usually* a one-to-one mapping of syntax tree constructs onto this type of boilerplate code. The API is relatively stable and robust, and constitutes the core of the LLVM C API.[2]

8.2.1 Reference Counting

As per the requirements of the language, the compiler needs to inject some sort of reference counting mechanism where necessary throughout the code. As I wrote earlier, destructors are automatically generated to deal with this problem (if a structure contains counted pointers, their reference counts need to be decremented when the structure falls out of scope or has itself reached refcount 0). But how does the compiler know when a variable needs to have its reference count incremented or decremented?

The case of assignment is an easy one; whenever a pointer variable is assigned, the old value is decremented and the new value is incremented. The more interesting case is when a variable falls out of scope. The compiler already has a table of symbols associated with the current scope. The variable manager module allows a callback function to be attached to a variable. When a scope frame is popped off the stack, these callbacks are run. This method allows the compiler to create counted variables with a callback function that generates decrement code; the callback is assigned at the time of creation. The compiler can then “forget” about the reference counting because the callbacks are run automatically.

This simple trick of assigning “fall out of scope” callbacks to variables manages to take care of a surprising number of cases—scopes are the most common reason to need to increment or decrement a counted variable. An added bonus of

[2] The documentation at http://llvm.org/docs/doxygen/html/group__LLVMC.html is extremely helpful

this system is that scope callbacks are only run when *the containing scope* is popped off the stack; if a counted variable is declared in a function scope, it will not have an increment/decrement cycle for every sub scope (loops or if statements, for example) inside the function. It would be incremented once at creation, and once at the end of the function.

There are some edge cases that make this system messy. When returning early from a function, it is necessary to run these callbacks for all current scopes before they are actually popped off. Furthermore, when dealing with generators, the implicit boxing of variables makes the reference counting through scope management break. As such, code in generators needs to be scanned to remove the reference counting manipulations caused by pushing and popping scopes.

Another interesting case is that of functions with a counted return type. The value returned from the function needs to be handled properly in two different scenarios: the returned value can be ignored or used. Both of these cases require reference counts to be managed separately. These values are considered "transient" by the compiler. They are put into a list and, if at the end of the compilation step for a statement which contains a function call, the value *is still in the transient list*, it is decremented. Otherwise it is left alone. These are known as "loaded transients" in the compiler, because they are returned with reference count one more than they would otherwise be. If the value is kept, the reference count remains the same (it is not incremented) and if it is not kept, it is decremented.

Despite the apparent complexity and numerous edge cases, the reference counting system works well and manages to produce code that does not let memory leak.

8.3 Step One: Prototype Generation

Unlike C, Eagle allows global names to be referenced before they are declared, without the need for a pre-declaration. As such, all names must be in the *LLVM scope* at the time of compilation, or else LLVM will crash with an error. In other words, the LLVM API needs to know about all function names and types before they can be referenced in a function call. If the Eagle compiler simply compiled functions in order, function prototypes like those in C would be necessary. Rather, the compiler adds all the function definitions to the LLVM IR code and *then* fills in the body of those functions afterward.

Function names are simply added to the LLVM code module—they have a complete calling signature but they lack a function body. LLVM does not care if a function has code attached to it; if, at the end of compilation, the body for a function is not found, it is assumed to be in a different object file and is therefore left to the linker to sort out. In this first step, all function names are added at the same time *before any actual instruction generation* so that during deeper AST traversal, all names are globally visible to LLVM.

Likewise, structure definitions are also registered with LLVM before actual code generation begins. Struct definitions have an additional step, however. Since structures may contain reference-counted pointers, they need to have implicit constructors and destructors to handle the memory management semantics. The programmer has no control over these functions and so they are generated during this first pass. The compiler will later inject calls to the constructors and destructors when creating or destroying a structure variable.

During this first pass, the compiler will also generate the data-portion of class definitions. The compilation of class methods is postponed until a later stage, but the space requirements of class instances needs to be known early on. Interface definitions are ignored by the compiler—an interface does not translate to any particular LLVM element. These definitions are used rather to build virtual method lookup tables for classes during method compilation.

8.4 Step Two: Class Method Compilation

Methods are compiled as functions whose first parameter is a pointer to the owning object. The compiler injects an implicit "self" variable and assigns it behind the scenes. Methods are otherwise indistinguishable from functions inside the compiler.

If a class implements an interface, method names are scanned for names that match the interface. If a matching method is found, its signature is verified and pointers are added to a statically-allocated virtual table. This table is generated at the time of method compilation; all new objects created for such a class are instantiated with their first structure member as a pointer to this virtual table. That table, along with runtime support described in a later chapter, provides dynamic method dispatch that interface polymorphism requires.

The user-defined initialization function is registered with LLVM and the compiler at this time. This function may not be called directly but can accept parameters as required via the `new` keyword syntax. Likewise, user-defined destructors may be defined but the programmer can not call them at any point. As part of the reference-counted scheme, every counted pointer can define a custom destruction function. The Eagle compiler will create an automatic destructor for every class, and inside of this destructor, it will inject a call to the user-defined destruction function as necessary.

A (simplified) example of this code generation process is shown in Figure 8.2. Both sides will produce equivalent code. After classes have been compiled, function code generation can begin. This setp focuses on simple procedures and generators.

```
class Greeter
{
    byte* name
    init(byte* name)
    {
        self.name = name
    }

    greet()
    {
        if self.name
            printf('Hello, %s!',
                self.name)
        else
            printf('Hello, World!')
    }
}
```

(a) A basic Eagle class

```
struct Greeter
{
    byte* name
}

func __egl_i_Greeter
(Greeter^ self, byte* name)
{
    self.name = name
}

func Greeter_greet(Greeter^ self)
{
    if self.name
        printf('Hello, %s!',
            self.name)
    else
        printf('Hello, World!')
}
```

(b) The struct-function equivalent

Figure 8.2: Method compilation equivalencies

8.5 Step Three: Independent Procedure Generation

Eagle has support for several types of independent code procedures (basically any code that is not contained in a class). These procedure types include functions, generators, and closures. This section will review the ways in which these types of procedures are compiled and will highlight some of the quirks associated with them.

Functions are the simplest top-level code block available. They follow the process outlined in section 8.2 precisely. There is nothing special about function code generation and the compiler does not perform any trickery to get them to work as they are defined in the specification. Closures and generators are the more interesting cases.

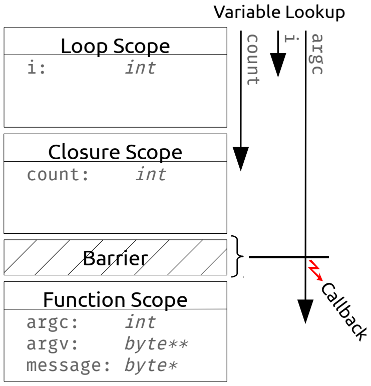
8.5.1 Closures

The compilation process of closures is relatively complicated, but it follows the same general arc as any other code generation phase. Closures are treated as separate functions with an implicit context. All variables local to the closure *itself* are compiled as any other variable. But closures can also capture the state of the scope in which they are defined, and closure functions can survive past the lifetime of that scope. It is therefore necessary to perform some trickery to "capture" variables in the enclosing scope.

The variable management system allows a "barrier" scope to be pushed onto the stack.[3] This barrier is created with a callback function that is executed whenever a variable lookup crosses it. In other words, when searching for an identifier in the scope stack, if the compiler finds the variable on the other side of a barrier, it executes a callback function with the variable information. This tool is leveraged when dealing with closures. Before the code in a closure is compiled, a barrier is pushed onto the stack. Thus, any variables local to the closure are one one side of the barrier, and any variables in the surrounding scopes are on the other side. Figure 8.3 shows an example of these barriers.

```
1  func main(int argc, byte** argv)
2  {
3    var message = 'Hello, world!'
4    var closure = func() {
5      int count = 0
6      for int i = 0; i < argc; i += 1
7      {
8        puts message
9        count += 3
10     }
11   }
12 }
```

(a) Eagle code



(b) Scope stack / lookup

Figure 8.3: An example of variable resolution within a closure (compiling at Line 9)

When the closure callback is run, the variable is implicitly wrapped in a reference counted object. The compiler replaces all references to the variable throughout its scope with this object, and automatically unwraps the variable so that its usage is transparent when accessing the variable. Thus, the variable is "lifted" into a reference counted environment. The closure retains a strong reference to this variable, allowing the variable to live beyond its natural scope. A further benefit of lifting the variables into a the counted paradigm is that multiple closures can reference the same captured variable; that variable will not be deallocated until the final closure is freed—the variable will never be invalid.

8.5.2 Generators

Generators are similar to closures in that they manage their own context via implicit structures. When compiling generators, the compiler produces code as it

[3] See src/compiler/variable_manager.c

normally would for any function, but at the end it scans the generated code and replaces any variable stack allocations with references into an automatically-created structure. All references to these variables are thus replaced. This context also stores a pointer to the current block of memory that is being executed; every yield statement is positioned at a block junction, and when the generator yields a value, a pointer to the next block is inserted into the context. The next time the generator runs, it executes a jump to the pointer referenced inside the context. Thus, it appears as if the generator yields control to the caller, when in actuality it does a full return. When the caller "returns" control to the generator, the generator code actually rebuilds the state of the function based on the context.

Chapter 9

Infrastructure

The compiler would not be complete without a hefty amount of infrastructure. There are both compiler-specific elements required as well as more general containers and data structures. Since the compiler is written in pure C, there is not a lot of library support. This is both a blessing and a curse; the compiler can tailor its implementations to the specific tasks at hand, but there is also a lot of boilerplate code involved in these basic modules. This chapter will highlight the most critical pieces of infrastructure necessary for the compiler to operate smoothly.

9.1 Handling Multiple Files

Inside the compiler it is necessary to both handle multiple files in separate logical code units (i.e. compiling multiple files at the same time), as well as deal with multiple files *referenced* by a single logical code unit (through code imports and exports). Both of these capabilities are required to make the compiler robust and usable in large projects.

The Eagle compiler uses GNU Flex to break down text input. Flex operates on individual files, but as per the specification, included code needs to be auto-generated into headers that are included in the current compilation unit. The compiler therefore needs to be able to lex multiple chunks of string data and file data all within the same logical unit. Fortunately, Flex provides a way to specify a custom buffering method, abstracting the notion of a file. The compiler uses this capability to seamlessly chain together strings and files for reading by the lexer and parser. This system allows for an extremely flexible parsing phase, giving rise to features such as the `--code` command-line option to specify extra chunks of code to compile.

On the other end of the equation, multiple logical compilation units can be compiled at compiler runtime. As a general rule, the Eagle compiler avoids global state. There are unavoidable instances however, and neither Flex nor Bison are thread-safe. Thus, when compiling multiple files, LLVM IR generation happens in a sequential manner. Fortunately this translation phase is relatively fast compared to the machine-code generation phase. After the compiler determines which command-line options are code files, it begins the code generation process on each one, one after another. The resultant LLVM code is stored in a list. Once all of these modules are converted to LLVM IR, the actual assembly generation process can begin.

9.2 Optimization, Assembly & Executable Generation

The LLVM library handles optimization and assembly code generation. Executables are created by invoking the system assembler and linker. In the case of the Eagle compiler, the process is simplified by invoking GCC after the assembly code has been created. The compiler stores assembly code and object files in the system temporary directory (`/tmp` on Linux), and the files are deleted at the end of the build process.

The LLVM framework is built with multithreading in mind. Since executable generation is the slowest step, it is very beneficial to split this process into multiple threads. The Eagle compiler allows for transparent multithreading—if the system does not support threads, the compiler falls back to sequential execution through a series of `\#ifdef ... \#define` and `typedef` declarations in the executable generation files.[1] The threads share a work queue containing the IR modules created by walking the syntax tree. The amount of work is therefore roughly equal across all threads, irrespective of module size. For large numbers of code files, the threading infrastructure can increase compilation times by two to three times, making the Eagle compiler satisfyingly fast.

9.3 Memory Management

As the Eagle compiler is written in C and not Eagle, it is necessary to manage memory manually in all cases.

For most short-term allocations, memory is freed as soon as it is done being used. In many cases functions need to return strings representing automatically-generated method names or type names. Many objects need to stay alive for a long period of time, however. For these objects (the AST tree, for example), the compiler uses a memory pool to collect objects to be freed. These pools are drained between each source file compilation step. Due to the fact that memory is often not freed immediately after it is used, the compiler is somewhat memory intensive. This consequence is compounded by the multi-pass process needed to fully scope all names and types. The memory intensity makes the compiler likely unsuitable for very small, embedded systems. This should not be a severe limitation, however, as the LLVM is very good at cross-compiling and targeting different architectures and systems (“Getting Started” (LLVM), *Hardware*).

[1] See `src/core/threading.c` and `src/core/shipping.c`

9.4 Containers & Misc. Libraries

The compiler can only tick because of a few essential components that are used extensively throughout the build process. As C does not have a robust standard container library, many of these tools needed to be written by hand.

9.4.1 Lists and Tables

Lists are built using array buffers. This container has proved absolutely indispensable for this project. Everything from the type system to the overarching structure to the code generation process relies on this code.[2] Array lists were chosen as the preferred list type as they have very nice performance characteristics: they are fast when appending items and they have extremely fast lookup. In general, because the compiler is deterministic during code generation (we do not often need to remove things from lists), many of the negative aspects of array lists are negligible in practice.

Tables are another key element and are particularly critical to the type system and variable management infrastructure. The tables in the compiler are implemented as *generic* hash tables.[3] Any hashing function can be provided, so it is possible to use these tables with a wide variety of objects. The table has an allocated buffer for "buckets," and within each bucket a linked-list is used for storing conflicting hashes. These tables resize as necessary and are very fast and efficient.

9.4.2 Regular Expressions

Regular expressions are not used as widely in the compiler; rather they are user-facing. The programmer can define which symbols to export using regular expressions. In order to parse and evaluate those regular expressions, the Eagle compiler uses a regular expression library designed to build trees and evaluate strings by walking the trees. These regular expressions are not quite as robust as those found in other libraries and languages, but they are significantly faster in some cases because they do not require backtracking.[4]

[2] See `src/core/arraylist.c`

[3] See `src/core/hashtable.c`

[4] See `src/core/regex.c`. The inspiration for this regular expression code comes from an article at https://swtch.com/ rsc/regexp/regexp1.html.

Part III

Conclusion

Chapter 10

A Non-Trivial Example

As part of the proof that this language is viable, I have created a program that plots mathematical functions. This project encompasses fourteen code files totalling over two thousand lines of Eagle code. The project is hosted on Github.[1] The plotter demonstrates several of the most important aspects of the Eagle language:

1. **Speed**: The plotter parses complex mathematical equations and calculates the output values at each window coordinate. The program can also calculate and render tangents on the fly, dynamically allocating and freeing the memory deterministically in real time, using the compiler-injected reference counting.

2. **C Interoperability**: The program uses the *Simple DirectMedia Layer* (SDL) library for displaying graphics.[2] SDL is a library written purely in C. Eagle is fully binary compatible with C, so calling SDL functions is as native in the program as calling functions written in Eagle, even when those functions deal with complex data types like structures.

3. **Object Orientation**: The program makes use of all of the object-oriented conveniences provided by Eagle. Interfaces are used to define an `Expression` type that provides an `eval()` method. The tree built up from the mathematical equations can thus be elegantly evaluated. Closures are used to iterate over array objects and views are used to allow range-based looping. The object-oriented code integrates seamlessly into the procedural code and does not make the procedural code look second-class.

4. **Dependencies**: The code is spread across fourteen files, each of which includes many other files. There are circular dependencies and multi-level dependencies, all of which are managed by the compiler to do away with

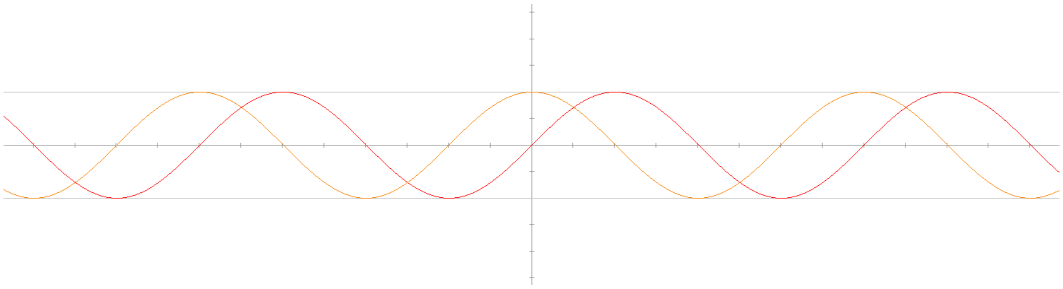
[1] http://github.com/samhorlbeck/plot-eagle

[2] https://www.libsdl.org/

the need for header files. Code is contained in different directories, so imported code files need to be resolved differently depending on the location of the importing file.

5. **Domains**: Several different tasks are accomplished in the code: low-level string manipulations are done in Eagle code, as well as lexing and parsing the input. Heavy computation is also done when calculating the floating point values for each of the time steps in the functions. On the other hand, the program cleanly implements a dynamic user interface, hiding the implementation details of that interface so other parts of the code can control aspects of the user experience in a simple, coherent manner.

Programming this medium-scale example in Eagle was a very edifying experience. While there remain bugs in the compiler, there were only a few occasions where I needed to apply my knowledge of those bugs to avoid problems with the plotting program. Below is a screenshot of the plotting program in action.



Chapter 11

What Now?

The Eagle language and compiler remain very much a work in progress. This chapter will enumerate what is working, what remains to be done, and future plans for the project.

11.1 What's Working

Both the language and the compiler remain in a heavy state of flux. The semantics and syntax of the language have begun to settle, with occasional large changes to reflect changing goals and design objectives.

11.1.1 Language Specifics

The language has a working (and somewhat robust) type system. Most of the basic C data types are implemented. Enhancements to the C suite of types are also present: classes and interfaces are defined, and their semantics and usage are quite formalized. The language also contains specifications for various other convenience features like closures and generators. The interface between code files via importing and exporting is well-defined, as are the basic control-flow structures.

11.1.2 Compiler

The compiler is becoming ever more robust as well. The basic elements, including parsing, code generation, assembly and executable generation, are all solidly in-place. The compiler can handle multiple files and supports outputting LLVM IR code, assembly files, object files, and executables. The compiler is *relatively* stable and will not crash with malformed input or syntax.

The injected reference counting system is fully operational and generally bug-free. Error- and some warning-messages are implemented, and platform-agnostic multithreading is stable. The language and the compiler have evolved together, so every construct defined in the language specification is fully supported by the compiler. The compiler is also relatively stand-alone, so pre-built binaries can be installed on systems without the LLVM framework already installed.

11.2 What's Left

There is still much work to be done on both the language and the compiler. This project will not end with the publishing of this paper.

11.2.1 Language Specifics

The language, while relatively mature, still has some gaping holes. Static global variables still have not been fully specified, for example. The semantics of type casting are confusing and need to be cleared up. Furthermore, several crucial design decisions still need to be made: will Eagle support operator overloading? How should macros and code generation tools be added, if at all? Many of the standard operators found in C-like languages are missing (there are no bitwise operators, for example).

It would also be useful to define the ways that generators could work in a closure or class context. Other object-oriented features, like views, could be brought to a more general setting rather than being tied to class definitions. Type extensions could also be added to allow traditionally non-object-oriented types to have methods.

Another key piece missing from the language is generics. Perhaps it is enough to have interfaces. The semantics of generics would be tightly tied to my ability to implement them in the compiler, so it is a difficult subject to address from the language-design standpoint. Generics are a double-edged sword, and ultimately C gets away with not having any form of generics. Such a change in the language specification would require much more thought and research, and perhaps community input.

Finally it might be nice to have some sort of basic language-supported multithreading model. Due to the nature of the reference counting, it would be difficult to require programmers to implement threads themselves using system libraries—the reference counting code is decidedly thread-unsafe, and it is injected by the compiler so the programmer has no power over it save not to use it. The goal of the language is to be as fast and simple as C, while also providing some convenience features that remove some of the boilerplate code. Language-supported threading would certainly fall in that category.

11.2.2 Compiler

There remain many bugs in the compiler that need to be addressed. They tend to be relegated to edge cases in the language, but their frequency is still unacceptably high for professional-quality code generation (in other words a corporate software team would not be pleased with the number of bugs inserted into output executables due to problems with the compiler).

It would be nice to have debug information (in DWARF format, for example)[1] included in the output executable. At the moment, the LLVM framework does have APIs for building debug information, but none of those APIs are exposed in the C interface, and the C++ APIs are extremely unstable.

Errors could be more coherent and there are more cases where warnings could be used to increase safety. Additionally, certain language constructs are not fully implemented; static variables, for instance, are not totally finished.

11.3 Release Timeline & Long-Term Goals

I plan to publicly announce this project in the coming months, as soon as some of the more fundamental missing features are added. Eventually I would like to add a basic standard library to augment the C standard library.

The Eagle compiler is free and open-source. The ultimate goal is to see it develop a thriving community of contributors and users. Language stability and a robust compiler will go a long way towards achieving these goals.

[1] DWARF is a standardized executable debugging information format used widely on UNIX-like systems.

Chapter 12

Final Thoughts

The Eagle language encourages a clean programming style for both high-level and low-level tasks. It does not make many guarantees about safety, but ultimately those guarantees are not necessary for the class of problems that Eagle is designed to solve. It can thus be both powerful for experienced users while remaining friendly for newer programmers. It integrates very nicely into existing environments due to its nearly-complete compatibility with C. It represents the modernization of C-like languages without removing that which is quintessentially C: the sheer power and simplicity of expression even at the lowest levels.

Eagle is not intended to be an academic curiosity or an expansion of existing theory. Rather it aims to be a pragmatic step in the evolution of programming languages, keeping an eye on the past while continuing to move forward. Its principles of speed, interoperability, and *developer responsibility* for dangerous constructs set it apart from many of its contemporaries. It fills a niche long-held by C, and continues the long legacy of the C mentality.

Chapter 13

Example Code

13.1 Basic “Hello World”

```
extern func printf(byte* ...) : int

func main(int argc, byte** argv) : int
{
    var message = ’World’
    if argc > 1
        message = argv[1]

    printf(’Hello, %s\n’, message)
    return 0
}
```

There is nothing particularly special about this example. It demonstrates the close visual similarities with C, at least on the surface. The `var` keyword is used to infer type from assignment. Notice the `if` statement does not require brackets. Eagle does allow one-line conditions.

13.2 Object Orientation & Closures

```
interface List {
    func add(any*)
    func get(int) : any*
    func each((any*:)^)
}

struct Node {
    Node^ next
    any* val
}

class LinkedList(List) {
    Node^ head

    func add(any* item) {
        var h
        if !self.head
            h = new Node
        else {
            for var h = self.head; h.next; h = h.next { 0; }
            h.next = new Node
            h = h.next
        }

        h.val = item
    }

    func get(int i) : any* {
        var h = self.head
        for int j = 0; j < i; j += 1 {
            h = h.next
        }

        return h.val
    }

    func each((any* :)^ callback) {
        for var h = self.head; h; h = h.next
        {
            callback(h.val)
        }
    }
}
```

```
extern func printf(byte* ...) : int

func main() {
    var list = new LinkedList()
    list.add(’Hello’)
    list.add(’World’)

    printEachAsString(list)
}

func printEachAsString(List^ list) {
    int count = 0
    list.each(func(any* item) {
        count += 1
        printf(’Item %d:\t%s\n’, count, item)
    })
}
```

In this example several aspects of the language are demonstrated. First, we define an interface called *List*, which defines a set of functions all objects adhering to *List* must follow. We then define a simple structure to contain our data and a class to operate on that data. All of the interface methods for *List* are defined in the class.

We also have an example of class instantiation. We could have defined an init function for the class, but that was not necessary in this case (the counted `head` node is initialized to `nil` automatically). In `main`, we use the list as a member of the *LinkedList* class. Later, we pass it to the `printEachAsString` function, which will accept any object conforming to the *List* interface.

This function contains an example of a closure. It captures the `count` variable and is called for every value in the list, as per the class definition. `count` will be incremented during every iteration as it is captured by reference, not by value.

13.3 Generators

```
gen fibonnacci(int n) : long
{
    long a = 1
    long b = 1

    for int i = 0; i < n; i += 1
    {
        long c = a
        long t = b
        b += a
        a = t

        yield c
    }
}

func main()
{
    for long i in fibonnacci(50)
    {
        puts i
    }
}
```

In this example we see how generators can be used in conjunction with range-based for loops. Generators do have a type (in this case it is `(gen:long)^`), so they can be prepared through the function call and then subsequently passed around; they need not be tied to a loop.

Bibliography

[1] Effective Go. *The Go Programming Language* https://golang.org/doc/effective_go.html, accessed February 2016.

[2] P. Fischer. Moore's Law and its Direct Impact on Software. 2015. *Intel Software and Services*. Intel. https://blogs.intel.com/evangelists/2015/04/18/moores-law-and-its-direct-impact-on-software/, accessed March 2016.

[3] Frequently Asked Questions. *The Go Programming Language* https://golang.org/doc/faq, accessed February 2016.

[4] cgo. *The Go Programming Language* https://golang.org/cmd/cgo/, accessed February 2016

[5] S. Johnson. Yacc: Yet Another Compiler-Compiler. 1975. http://dinosaur.compilertools.net/yacc/, accessed December 2015.

[6] S. Kell. In Search of Types. *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 227–241, 2014. http://doi.acm.org/10.1145/2661136.2661154, accessed December 2015.

[7] B. Kerningham & D. Ritchie. The C Programming Language. *Prentice Hall*, 2nd edition, 1988.

[8] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. *University of Illinois, Thesis*, 2002. http://llvm.org/pubs/2002-12-LattnerMSThesis.pdf, accessed December 2015.

[9] C. Lattner & V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 75–, 2004. http://dl.acm.org/citation.cfm?id=977395.977673, accessed December 2015.

[10] LLVM Developer Policy *LLVM Documentation* http://llvm.org/docs/DeveloperPolicy.html, accessed March 2016.

[11] Getting Started *LLVM Documentation*
http://llvm.org/docs/GettingStarted.html, accessed March 2016.

[12] mog T. Mogensen Basics of Compiler Design *University of Copenhagen*, Department of Computer Science. lulu.com, 2010.
http://www.diku.dk/ torbenm/Basics/basics_lulu2.pdf, accessed March 2016.

[13] D. Ritchie. The Development of the C Language. *The Second ACM SIGPLAN Conference on History of Programming Languages*, 201–208, 1993.
https://www.bell-labs.com/usr/dmr/www/chist.html, accessed December 2015.

[14] The Rust Programming Language. *Rust Documentation*
https://doc.rust-lang.org/book/, accessed February 2016.

[15] J. Sammet. Programming Languages: History and Future. *Commun. ACM*, 601–610, 1972. http://doi.acm.org/10.1145/361454.361485, accessed December 2015.

[16] M. Lesk & E. Schmidt. Lex – A Lexical Analyzer Generator. *UNIX Vol. II*, 375–387, 1990. http://dinosaur.compilertools.net/lex/, accessed December 2015.

[17] The Swift Programming Language (Swift 2.2) *iOS Developer Library – Prerelease*
https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/index.html, accessed March 2016.

[18] toal R. Toal. Programming Paradigms *Loyola Marymount University Website*
http://cs.lmu.edu/ ray/notes/paradigms/, accessed March 2016.