Designing a Compiler for a Statically-Typed Language

Ethan White and Jonathan Huo
March 1st, 2016, to May 20th, 2016

Contents - 1

Acknowledgements - 2

Thank you to Kevin Watt for providing the workspace and resources from where the research for this paper was conducted.

Introduction - 3

In the days when dinosaurs roamed the earth, and the year matched the regular expression /195[0-9]/, one would write in the native (binary) instruction language of the computer that is running the computation, often on punched cards, without any abstraction whatsoever.

However, it was quickly realized that this was unsustainable; programmers are only humans, after all, and it would be much nicer if machines would assist them in writing software. People began using *assembly languages*, which, although simply mnemonics for what the programmer would be writing on punched-cards or directly in binary, allowed one's code to be much more readable and maintainable. Assembly is still occasionally in use; modern assembly looks somewhat like:

```
   SECTION .data           ; data section
msg:    db "Hello World",10     ; the string to print, 10=cr
len:    equ $-msg               ; "$" means "here"
                                ; len is a value, not an address

        SECTION .text           ; code section
        global main             ; make label available to linker
main:                           ; standard  gcc  entry point

        mov     edx,len         ; arg3, length of string to print
        mov     ecx,msg         ; arg2, pointer to string
        mov     ebx,1           ; arg1, where to write, screen
        mov     eax,4           ; write sysout command to int 80 hex
        int     0x80            ; interrupt 80 hex, call kernel

        mov     ebx,0           ; exit code, 0=normal
        mov     eax,1           ; exit command to kernel
        int     0x80            ; interrupt 80 hex, call kernel
```

Later, we developed higher-level languages; they were further abstractions upon assembly languages, and introduced concepts such as *variables*, *structs*, *methods*, *classes*, and others. We saw the development of simple automatic memory management; when a variable goes out of scope, its memory is freed. Later, we even saw automatic reference counting. The ever-famous C programming language is one of the earlier higher-level languages, and was developed in the seventies (the veritable heyday of computing innovation) by a lone computer nerd working at Bell Laboratories - Dennis Ritchie. An example:

```
int main() {
        int one = 1;
        int two = 2;
        int product = one * two;
        printf("A number: %s", product);
}
```

But these were all essentially elegant, human-readable ways to represent the native instruction set of the CPU. No matter how complicated things seem, every single language construct in C can be compiled down to some form of native CPU assembly. One major innovation came in the form of the Lisp programming language. No longer were you simply writing binary in disguise; instead, you were writing for an idealized virtual machine, that was run on top of the underlying hardware. This enabled new features that would be simply impossible when writing a low-level language, such as garbage collection, dynamic typing, and reflection.

Today, we see C-style languages run on virtual machines everywhere, with a prominent example being the Java programming language. It benefits from the performance improvements afforded by a just-in-time compiler, or JIT, as well as the usability benefits of a statically-typed language.

This Independent Study was an attempt to create a statically-typed language that, like Java, was compiled down to bytecode that ran on top of a virtual machine, and with an aim to incorporate novel features and step back to ask what we can improve over traditional languages.

Designing a Compiler for a Statically-Typed Language - 4

Background - 4.1

A Brief Introduction to Compilers - 4.1.1

Most high-level programmers have little knowledge regarding the interior schematics of a compiler. All they know, and normally only care about, is that a compiler program translates the code that they wrote in their language of preference, to some other language that's machine-readable. And in a sense, that is essentially what compilers are - translators between programming languages. A good analogy to help understand the abstract concept of compilers would be to compare compiler programs to file conversion programs - specifically JPEG to PNG image converters. Image converters reformat the data used to describe pictures while maintaining the same visual display; compilers take code written in one programming language and output code in another programming language, while maintaining that the outputted code possesses the same meaning as the inputted code. A common misconception is that compilers can only translate high-level languages to low-level languages. It's not an inherently true statement, considering the fact that compilers that deal with the conversion between languages like Java and C# do exist (albeit, those two languages are almost identical), but it does point out what compilers are generally used for - turning high-level, human-readable code, into easily processable, fast-to-execute, machine-readable code. In the present case, however, this generalization certainly applies, since my compiler deals with the conversion of the high-level Slang Programming Language into its low-level counterpart, Slang Bytecode.

The process of compiling a language is broken up into several unique stages. In compiler design, these parts fall into one of two categories: the frontend phases and the backend phases. The frontend phases are responsible for analysing the input text (the code received as input) and storing the analysed data in a logical data structure. This data structure holds important information about the semantics (meaning) of the input, and is consequently known as the semantic representation. The backend phases then read and synthesize the information held by the semantic representation to produce output text in the target language (code generation). This paradigm of analysis, representation, and synthesis is incredibly useful because of its modularization. Since the frontend and backend components can only communicate through the semantic representation, these two modules can be developed independently, enabling programmers to develop frontends for different source languages and backends for different target languages, as long as the semantic representation follows a consistent interface.

Linguistic Theory - 4.1.2

Linguistic theory is the basis of all language applications. As the formal study of grammatical construction (syntax), it describes the fundamental theoretical principles and algorithms that compilers need to process input text. In the following sections, we will delve deep into the process compilers use to generate syntax trees, the theory behind it, and the implementation.

Tokenization - 4.1.2.1

As the first stage in compilation, tokenization (lexical analysis) processes the raw input and prepares it for parsing. It's the process used to determine whether a group of characters or symbols form a valid word (token). A tokenizer (program that performs tokenization) analyses the input and produce a buffered stream of tokens for the parser to use, rather than a raw input string. It does the important job of stripping away whitespaces, comments, unnecessary annotations, and categorizing tokens into groups based on their semantic value (meaning). This method of symbol recognition that compilers use is very similar to the way humans process English words. At the lowest level, English sentences can be represented as an ordered collection of characters and symbols; of which there is a finite set, known as the alphabet. Individually, each character in the alphabet has no meaning, but when combined, these characters form words. When reading sentences, humans are able to piece together characters to match words; of which there is another finite set. In many natural languages (languages used by humans), a formalized list of these words can be found in a dictionary - a map between words and their meanings. Similarly, tokenizer implementations scan through each character in the input and try to match patterns of them into words, which are then categorized (think nouns, verbs, and adjectives) based on their meanings.

Parsing - 4.1.2.2

After the tokenization stage is complete, the initial raw input becomes prepared for parsing (syntax analysis). At this step, the parser (program that performs parsing) analyses the token stream produced by the tokenizer, and generates an internal representation of the semantics of the input code. In most situations, this representation takes the form of an abstract syntax tree (AST).

Much akin to the tokenizer, the parser borrows most of its strategy from human reading patterns - specifically the method we use to associate nouns with verbs and adjectives. To exemplify this point, let's examine the following English sentence by hand:

Mark saw Jeff.

When reading such a sentence, humans try to group together related terms. For instance, our minds attach the words "Mark" and "saw" together, because the verb "saw" is used to refer to an action being performed by a subject, which in this case, is Mark. Furthermore, the word "saw" also implies that Mark is performing the action upon another subject, which in this case, would be "Jeff". We can then simplify and generalize these three words by combining them into one category - an operation (since the word "saw" implies that some entity performed some action upon another entity). In linguistic theory, this type of categorization is known as a production rule; a state in the parser that dictates when to combine a specific pattern of tokens into a more generalized form for easier processing. This is where the parse tree comes into play. Linguists and computer scientists use this specialized form of tree to represent where tokens match production rules. Each node in the parse tree is an embodiment of a production rule matched by the parser, and the act of constructing a parse tree is known as parsing. To illustrate; a parse tree generated for the sample sentence:

- *sentence*
    - *subject*
        - *action*
            - *pronoun*
                - *"Mark"*
            - *verb*
                - *"saw"*
            - *pronoun*
                - *"Jeff"*
    - *punctuation*
        - *period*
            - *"."*

Observing the above parse tree, we can see that the tokens "Mark", "saw", and "Jeff" matched the production rule *action*. In formal notation, such production rules are defined like so:

rule → TOKEN TOKEN
TOKEN → "hello" | "goodbye"

In this case, linguists will say that the rule *rule* is the production of two *TOKEN*s in a row, and the rule *TOKEN* is the production of the strings "hello" or "goodbye" (logical and set operators are defined using regular expression notation). A string is said to match a production rule if it matches the regular expression used to define the rule - this regular expression is defined on the right-hand side of the arrow (→), and may contain other production rules. Any set of production rules that use each other is known as a grammar; these are used to describe languages in which text can be written. A rule whose definition does not contain any subrules is known as a terminal symbol; in vice-versa, a rule containing subrules is known as a non-terminal symbol. In the above example, *rule* is a non-terminal symbol, and *TOKEN* is a terminal symbol. The rule at the top of the rule hierarchy is called the start symbol, and its derivation (more on that later) must encompass all other rules. For now, it will suffice to say that the start symbol will always be the root node of the parse tree.

We are now able to deepen our understanding of parsers by exploring the concept of derivation and its role in the creation of parse trees. Consider the following input, parse tree, and the grammar used to generate it:

*1 * 1 + 2*

The AST:

```
⤷ expr
    ⤷ expr
        ⤷ INT
            ⤷ "2"
    ⤷ PLUS
        ⤷ "+"
    ⤷ expr
        ⤷ expr
            ⤷ INT
                ⤷ "1"
        ⤷ TIMES
            ⤷ "*"
        ⤷ expr
            ⤷ INT
                ⤷ "1"
```

And the grammar (repeated definitions of a production rule represent alternative ways to match the rule):

INT → [0-9]+
PLUS → "+"
TIMES → ""*
expr → INT
expr → expr PLUS expr
expr → expr TIMES expr

We can perform a derivation of the input string by applying a sequence of production rules to transform the start symbol back into the input string. This is important, because it allows us to mathematically prove that our input string conforms to the grammatical rules of our language, which is exactly what the parser is trying to accomplish. Beginning with the start symbol, a derivation of a string is performed by progressively expanding non-terminals until we are eventually left with the input string. As a demonstration, I have performed a derivation of the string "*1 * 1 + 2*" using the rules defined by our grammar. Each step in the derivation represents the rewrite of the rightmost non-terminal using one of the production rules:

1. *expr* (start symbol)
2. *expr TIMES expr* (rightmost non-terminal expanded with rule *expr*)
3. *expr TIMES expr PLUS expr* (rightmost non-terminal expanded with rule *expr*)
4. *expr TIMES expr PLUS INT* (rightmost non-terminal expanded with rule *expr*)
5. *expr TIMES expr PLUS 2* (rightmost non-terminal expanded with rule *INT*)
6. *expr TIMES expr + 2* (rightmost non-terminal expanded with rule *PLUS*)
7. *expr TIMES INT + 2* (rightmost non-terminal expanded with rule *expr*)
8. *expr TIMES 1 + 2* (rightmost non-terminal expanded with rule *INT*)
9. *expr * 1 + 2* (rightmost non-terminal expanded with rule *TIMES*)
10. *INT * 1 + 2* (rightmost non-terminal expanded with rule *expr*)
11. *1 * 1 + 2* (rightmost non-terminal expanded with rule *INT*)
12. *1 * 1 + 2* (input string)

The above derivation can be expressed visually by creating a parse tree that represents the process with which the parser derived the original input string. On each step of the derivation, the parser expands the rightmost non-terminal and rewrites it with a lower-level representation of the grammatical rule used for the expansion (at step 1, the leftmost non-terminal *expr* was expanded to *expr PLUS expr*, which is a lower-level and less generalized form of the rule *expr*). Beginning with the start symbol, we can represent each non-terminal as a node in the parse tree. Every time we expand such a node, we can append the nodes generated from the expansion to the node representing the non-terminal; thus creating a new level in the parse tree. To exemplify:

- *expr* (start symbol; appended during step 1)
    - *expr* (appended during the step 3 expansion)
        - *INT* (appended during the step 4 expansion)
            - *"2"* (appended during the step 5 expansion)
    - *PLUS* (appended during the step 3 expansion)
        - *"+"* (appended during the step 6 expansion)
    - *expr* (appended during the step 3 expansion)
        - *expr* (appended during the step 2 expansion)
            - *INT* (appended during the step 7 expansion)
                - *"1"* (appended during the step 8 expansion)
        - *TIMES* (appended during the step 2 expansion)
            - *"*"* (appended during the step 9 expansion)
        - *expr* (appended during the step 2 expansion)
            - *INT* (appended during the step 10 expansion)
                - *"1"* (appended during the step 11 expansion)

With this new knowledge in hand, we can make some more observations about parse trees. As mentioned before, parse trees have terminal symbols as the leaf nodes, non-terminals as the interior nodes, and the start symbol as the root node - this idea is emphasized by the concept of derivations. We also can see that an in-order traversal of the parse tree leaf nodes will yield the original input string. Most importantly, however, the parse tree shows the association and order of the operations, while the raw input string does not. Notice that in the parse tree above, the *TIMES* operation is a subtree of the *PLUS* operation. In order to compute the value of the *PLUS* tree, the value of the *TIMES* subtree must be obtained first (in linguistic terms, it is said that the *TIMES* is bound more closely than *PLUS*). This raises some questions: how did the parser determine that it should choose the derivation where *TIMES* is bound more closely than *PLUS*? With our sample grammar, there is no rule dictating that the parser should generate the tree in this fashion. An equally valid (but incorrect, according to the order of operations) derivation would be to bind *PLUS* more tightly than *TIMES*. This is resolved through dynamic derivation order. If we observe the process used in the sample derivation, we can see that on each step we are always unraveling the rightmost non-terminal - this mode of derivation is known as a rightmost derivation. Conversely, we can also perform a leftmost derivation; one where the leftmost non-terminal is expanded on each step. In the current parser, we are using rightmost derivation to discover and expand production rules. However, if the input string was rewritten like so: *1 + 1 * 1*, then the *PLUS* rule would be more tightly bound than *TIMES*, which would then require the parser to use leftmost derivation if we wanted to maintain order of operations. The resolution to this problem is interesting. Dynamic derivation allows the parser to hop around and select where to begin the next step in the derivation; rather than applying rules from one end to the other in a linear motion. With this, we are able to have the parser decide which section to parse next - enabling us to modify the bindings of each rule. However, it is very difficult to implement dynamic derivation in code; usually, if custom rule binding order is desired, the grammar is restructured, rather than the parser itself:

INT → [0-9]+
PLUS → "+"
TIMES → ""*
addition → multiplication PLUS multiplication
multiplication → INT TIMES INT

A grammar in the above form will always yield parse trees that have *multiplication* bound more closely than *addition*; however, parse trees formed using this grammar will contain extra *addition* nodes.

At this point, it is important for us to distinguish the difference between a parse tree and an abstract syntax tree (AST). The big separator here is that the parse tree is strictly a linguistic tool - it follows the derivation of an input string, and proves that it conforms to a grammar, but it doesn't provide much information regarding the structural and executable aspects of a program. An abstract syntax tree is much more adept at that; it's a reorganized form of the parse tree that contains specific details and annotations that make the compiler's task of code generation easier. For this reason, it is called an abstract syntax tree - it's a higher-level representation of the strict, concrete syntax tree, that's geared towards making compilation faster and more effective. As a comparison, we can examine an example parse tree (concrete syntax tree) and AST for this snippet of Python:

```
def print_word(word, suffix):
      print(word + suffix)
```

A concrete syntax tree for the above code would look somewhat like this:

- ↳ *function_definition*
    - ↳ *“def”*
    - ↳ *print_word*
    - ↳ *parameters*
        - ↳ *“word”*
        - ↳ *“,”*
        - ↳ *“suffix”*
    - ↳ *body*
        - ↳ *function_call*
            - ↳ *print*
            - ↳ *arguments*
                - ↳ *concatenation*
                    - *“word”*
                    - *“+”*
                    - *“suffix”*

An abstract syntax tree, however, would take a form like this (dashes indicate annotations; they are essentially key/value pairs):

- ↳ *program*
    - ↳ *function_definition*
        - name : *“print_word”*
        - ↳ *parameters*
            - ↳ *parameter*
                - name : “*word*”
            - ↳ *parameter*
                - name : “*suffix*”
        - ↳ *function_body*
            - ↳ *function_call*
                - name : “*print*”
                - ↳ *arguments*
                    - ↳ *argument*
                        - ↳ *expression*
                            - operation : “*concatenation*”
                            - ↳ *word*
                            - ↳ *suffix*

Observing the two trees, we can see that they have a few major differences. The parse tree strictly adheres to the derivation and offers little to no abstraction, while the AST provides an augmented and annotated form of the parse tree that's streamlined for easy code generation. The AST also compacts the tree by stripping away nodes that don't provide much semantic value and replaces them with meaningful annotations, such as single-successor nodes, and tokens like commas, parentheses, and redundant keywords.

We can now take a look at the different types of parser programs and the strategies that they employ to find and prove the derivation of an input string. The first and most straightforward parser that we'll be examining is the recursive descent parser. As one would expect, the parser reads the input in a left-to-right (and thus, recursive descent parses text through rightmost derivation) fashion, where terminals are seen in the order they appear in the token stream. The parse tree is constructed in a top-down manner; starting with the root node (start symbol) and ending with the leaf nodes (for this reason, recursive descent falls into a broad category of parsers known as top-down parsers). With recursive descent, we maintain three stacks at all times - a stack *R* to represent the set of processed symbols; a stack *U* to represent the set of unread tokens; and a stack *P* to represent the set of production rules in the grammar. At each step of the parse, we check if the pattern of tokens in *R* (or the rightmost set of tokens) matches, or has the potential to match, a production rule found in *P*. If there is a single production rule in *P*, and it is matched by the tokens in *R*, then we considered the token pattern in *R* to be valid derivation, and we append nodes to the parse tree corresponding to said derivation. We then refill *P* to include all of the rules in the grammar, and we pop the next token in *U* and push it to the top of *R* (in the future, we will denote the action of popping from *U* and pushing to *R* as a "read operation"). Otherwise, if the tokens in *R* do not currently match any production rule, but *P* still contains potentially matchable rules; we remove the set of rules that we know cannot be matched from *P*, and if there are still rules left in *P*, then we read another terminal from the stream. If there are no plausible rules left (i.e., the stack *P* is empty), then we know that the input string is not a valid derivation of our grammar's rules, and the parser can terminate with an error. Otherwise, the logical statements above are reevaluated in a continuous loop until a full parse tree is formed and there are no tokens left to read from *U*. However, if by the time the parser runs out of tokens to read there is still more than one rule left in *P*, the parser should match the rule whose definition contains the smallest, additive-number of terminals. As a demonstration of this parsing technique, consider the process used to parse following input string and grammar:

abc hello abc

And the grammar:

s → [z | q]+
z → q "hello"
q → "abc"

The main concept that needs to be understood here is the process in which the parser determines what rules are plausible. We can show how this is accomplished by walking through the parsing process. The double-pipe (||) denotes the separation of the sets *R* and *U*; tokens to the left of the pipe are in the stack *R*, and tokens to the right are in stack *U*:

1. *"abc" || "hello" "abc"*
   After reading the first token of input, "abc", we can see how *R* already has the potential to match two of the rules in our sample grammar - rule *z*, and rule *q*. Both these rules match strings that begin with the character sequence "abc". As of now, rule *q* is an absolute match of stack *R*, but we cannot confirm a match since there are still other plausible rules in *P*.

2. *"abc" "hello" || "abc"*
   Now that two tokens have been read, rule *q* no longer matches the token sequence in *R*; thus, we can pop *q* off of *P*. We are left with a single rule in *P* (rule *z*) that is an absolute match of the token pattern in stack *R*. Since this is the case, we can replace the tokens in the current *R* stack with name of its production rule, and we can also append the nodes corresponding to this derivation to the parse tree.

3. *z abc ||*
   At this point, the parser has processed all tokens in *U*. Now, the rightmost symbol on the stack *R* matches rule *q* and has the potential to match rule *z*. However, since there are no terminals left to read, we match the rule whose production rule contains the smallest additive-amount of terminals. In this case, such a rule is *q*.

4. *z q ||*
   Here, the parser can match rule *s* to the symbol pattern in the stack *R*. There are no other plausible productions, so we confirm a match for *s*.

5. *s ||*
   Finally, the parser reaches a stage where there are no terminals left to read (i.e. stack *U* is empty), and the set of production rules that match the symbols in *R* is empty. We halt, and return the parse tree as the output.

The generated parse tree:

- ↳ *s*
    - ↳ *z*
        - ↳ *q*
            - ↳ *"abc"*
        - ↳ *"hello"*
    - ↳ *q*
        - ↳ *"abc"*

Although recursive descent parsers are simple to understand and relatively quick to implement by hand, they aren't exactly the most versatile of all parsers. On every step of the parse, the recursive descent strategy compares each production rule to the stack, and backtracks if a match isn't found. This results in the parse taking exponential execution time when dealing with complex grammars. Now, there is a solution to this problem, and that is to parse the input using a subset of recursive descent parsers, known as the predictive parser, which runs in deterministic linear time. However, predictive parsers are simply not powerful enough to parse most complex grammars, due to the nature of the algorithm they use. Recursive descent has another inherent problem with its algorithm; their implementations often lead to jumbled, unmaintainable code. For these reasons, compiler programmers generally only choose recursive descent when working on parsers for simple grammars, and use a more powerful and maintainable table-based parser for parsing more complex grammars.

For complex grammars, compiler developers prefer to use the more powerful, table-based, bottom-up parsers over the top-down, recursive descent parsers. As one might expect from their names, bottom-up parsers perform the exact opposite set of operations that a top-down parser would. In top-down parsing, we validate input strings by performing a derivation through production rules, and generating the parse tree in a top-to-bottom fashion, starting from the root of the tree (start symbol). With bottom-up parsing, we start with the input string and reduce it into the start symbol through a series of reverse productions (these are known as "reductions"). Normally, a derivation would trace the transformation of the start symbol to the input string; with bottom-up parsing, we perform those same operations, except in a reversed manner. Naturally, the construction order of the parse tree is reversed as well - the parser combines small portions of parse trees together to create larger trees; eventually reaching the start symbol, or root node. To illustrate, we can use the bottom-up technique of performing reversed-rightmost derivations to parse previous sample grammar and input string (double pipe denotes separation of read and unread symbols; symbols to the left have not been read, symbols to the right have):

1. || *"abc" "hello" "abc"* (input string)
2. *"abc"* || *"hello" "abc"* (token read)
3. *q* || *"hello"* "*abc"* (terminal "abc" reduced with rule *q*)
4. *q* "*hello"* || *"abc"* (token read)
5. *z* || *"abc"* (non-terminal *q* and token "hello" reduced with rule *z*)
6. *z "abc"* || (token read)
7. *z q* || (terminal "abc" reduced with rule *q*)
8. *s* || (non-terminals *z* and *q* reduced with rule *s*)
9. *s* || (start symbol)

Notice how each operation we performed in the parse is either reading a terminal, or combining symbols via reductions. In parse theory, this strategy is known as shift-reduce parsing, and is the sole method used by parsers falling into the bottom-up category. Its name comes from the fact that in its implementation, there are only two operations that are ever performed on the two symbol stacks (the stack of processed symbols and the stack of unread tokens) - reading a terminal from the set of unread tokens, which is called shifting; and performing inverse productions on the right end of the set of processed symbols, which is called reducing. These operations are typically implemented through the stack data structure. Like top-down parsing, we can represent the set of processed symbols with a stack, where the most recently read symbols at the top. The set of unread terminals can also be represented as a stack, where the next token to be read is at the top. Whenever the parser shifts, we pop the first token on the unread stack and push it to the top of the processed stack; whenever the parser reduces, we pop the group of symbols that matched the reduction rule and replace them with the corresponding rule name.

Defining Languages - 4.1.2.3

Formal grammars can be defined by writing down their production rules. These production rules are written in the form of *rule_name → sequence_to_match*; where the left side of the arrow (→) represents the name of the rule, and the right side represents the symbol sequence that produces a match for said rule. However, often becomes tedious and unintuitive to define large, complex grammars using such a form. As of such, linguists and computer scientists greatly prefer to use more efficient formats to represent languages they define. Of these, Backus-Naur Form (BNF) is one of the more predominant scripts for grammar notation:

```
<statement> := <addition>+
<addition> := <multiplication> (‘+’ <multiplication>)*
<multiplication> := <int> (‘*’ <int>)*
<int> := [0-9]+
```

As you can see, BNF is a lot nicer to read than the traditional way of denoting the syntax of a language. Terminal strings are represented by enclosing the desired character sequence within single quotes, non-terminal rule definitions are defined by enclosing the desired name within a matching pair of angled brackets, and the production for these rules is defined on the left hand side of the colon-equals. Once again, logical and set operations are shown using regular expression notation.

Grammar notation scriptures are generally used to define production rules and other metadata for parser generation and development programs (parser generators), such as ANTLR, GOLD, YACC, and BISON. By reading the attributes of a grammar specified through BNF scripture, a parser generator program can produce code in any programming language for a parser using such a grammar. Parse trees generated by generated parsers come with their own runtime API, so the programmer developing the language can directly manipulate parser output.

Semantic Analysis - 4.1.3

Parsers are powerful programs that can detect errors in the syntax of a input text according to a specified grammar, and furthermore, generate an abstract syntax tree that shows the semantic relations between different parts of the input text. However, what they cannot do is handle the context of language segments based on rules in the execution of the program binaries - such as checking whether variables are properly defined; whether functions and values are used in accordance to their types; or whether referenced fields in a struct object actually exist. This is the task of the semantic analyser - a program that walks and validates the semantic context of the parse tree, checking for errors undetectable by the parser. We can show the difference between the function of these two programs with the following snippet of C code:

```
int i = 5;
```

```
char c = ‘c’ + i;
int e = v;
```

Clearly, there are some major errors with this program; for instance, the variable *i* is of type *int*, and thus cannot be added with a literal of type *char*. As well, we define *e* on the third line, except we set its value to be that of a non-existent variable *v*. Obviously, we can tell that this code will not compile - both a type error and a missing variable error will be thrown. However, to the parser, this is completely syntactically valid code. Nowhere, in any of these definitions, are we missing any sort of operator, type definition, semicolon, or whatnot. From this broken code, the parser can still apply all of its derivations and generate a full parse tree, since it cannot recognize when variables do not exist, when their usages do not conform to their defined type, or when they are used out of scope. Unless we introduce complicated context evaluation into our parser, it simply won’t be able to detect such mistakes. This is where analyser (program that performs semantic analysis) comes in. By walking the AST (or parse tree) generated by the parser, the analyser can check for these non-syntactical errors, such as variable and parameter misusage, and allows us to reject these inputs so that the compiler can proceed with code generation.

Static Scope and Type Checking - 4.1.3.1

In almost all modern compilers (with the exception of Lisp and some areas of JavaScript), the usage of variables, methods, classes, and structs are all restricted based on their scopes. Typically, a programmer is only allowed to make references to these data structures from enclosing scopes. To give an example of this, consider the following segment of invalidly-scoped C code:

```
int five = 5;
if (five == 5) {
      if (five < 6) {
            int six = 6;
      }
      five = six;
}
```

This snippet of code contains a classic example of using variables outside of the scope they were defined in. Variable *six* was declared in an *if* clause that was in a lower-level scope than variable *five*, which later tries to access *six* from a higher-level scope than the one that *six* was defined in; thus resulting in the compiler yielding a scope error (or, an undefined variable error, depending on the error handling capabilities of the compiler in question).

Static scope checking can be implemented through a bit of clever data structure manipulation and AST traversal. From a first glance, we can already tell some things about the implementation based on the hierarchical structure of the above snippet of C. Within the semantic analyser, we will need to keep track of the relative associations of the various scopes in the program. When visualized, these scopes form a tree-like data structure; in which scopes can contain both variables and other scopes, allowing there to be a hierarchy of nested scopes. Observe, a representation of the scoping system in the above C program:

```
SCOPE GLOBAL :
        contains :
                five : int
        nested :
                SCOPE IF :
                        contains :
                                six : int
                        nested :
                                SCOPE IF :
                                contains :
                                        empty
```

The immediate response for implementing this sort of data structure would be to have a scope class, and allow such a class to contain a set of variables, and a set of nested scopes. However, this model loses its efficiency when ASTs must be traversed more than once, because it then becomes difficult for the semantic analyser to find the correct pre-existing scope objects to map other objects to (this is often the case when implementing forward-referencing capabilities for functions). Observing the above tree, you may have realized that would be redundant to create another tree for the sake of scope-checking, when an already existing representation can perform the same task - the AST. Consider the following example:

- *program*
    - *variable_declaration*
        - name : "*abc*"
        - type : "*int*"
        - *expr*
            - *5*
    - *if_clause*

        …
        - *variable_declaration*
            - name : "*cba*"
            - type : "*int*"
            - *expr*
                - *addition*
                    - *1*
                    - *abc*

The AST is already a perfect representation of the scoping hierarchy - the semantic analyser just needs to make the information about these scopes more readily accessible. This can be done through adding a few extraneous fields to the class that represents the AST nodes; a reference to the current scope's parent scope, a hashset to hold the defined variables, and another hashset to hold the nested scopes. Below, an implementation in Java (somewhat akin to the format used to describe custom AST annotations in the ANTLR parser generator API):

```
        public class ASTNode {
                ... // the existing data used to describe nodes and etcetera

                private ASTAnnotation[] notes;
        }

        public class ASTAnnotation {
                ... // some data
        }

        public class ScopeAnnotation extends ASTAnnotation {
                private ASTNode parentScope;
                private HashSet<Variable> variables = new HashSet<Variable>(); // we
define the Variable class somewhere else, for the sake of simplicity.
                private HashSet<ASTNode> nestedScopes = new HashSet<ASTNode>();
        }
```

As the analyser traverses the AST, it can label key nodes with information regarding variable declarations and nested scopes. We can now reformat the previous AST with the newly added annotations feature:

- *program*
    - parentScope : "null"
    - variables :
        - *abc*
            - type : "*int*"
    - nestedScopes : "*if_clause"*
    - *variable_declaration*
        - name : "*abc*"
        - type : "*int*"
        - *expr*
            - *5*
    - *if_clause*
        - parentScope : "*program*"

          …

When traversing the AST, the analyser always maintains a pointer to the current scope, called *C*. Whenever a scope declarator node is visited (i.e. *program*, *if_clause*, *method_body*, *for_clause*), it sets the *parentScope* attribute of the node to be the value of *C*, and then we update *C* to be set to the said scope declarator. From thereon, any variable or nested scope it visits has a reference of itself added to its corresponding set within its enclosing scope (the newly defined value of *C*). Similarly, if the analyser exits a scope, we can update *C* to become the scope it entered, and continue the traversal from there.

Static type checking is a very common issue within the implementation of most mainstream programming languages, such as Java, C#, C, C++, and Go. In type systems such as these, where variables and functions are not allowed to have mutable type and return value, the semantic analyser must be able to check whether these data types are used accurately. This is very easy to implement, given our current architecture for semantic analysis. Whenever a variable's usage is encountered during AST traversal, simply check if its type matches the one mapped to its name in the annotation that describes the variables within the current scope.

Code Generation - 4.1.4

Code generation is the final stage in compilation, and barring any optimizations, is often the easiest to implement (in the case of our project, all code optimizations were done on the interpreter side). When dealing with conversion to lower-level, instruction oriented bytecode, compilers need only to worry about the correct generation of key instructions at this stage. High-level tasks, such as register management details and the whatnot are all usually dealt with during semantic analysis. To give a sense of what bytecode feels like, we can generate machine instructions for the arithmetic expression *1 + 1 * 2* (this is not a comprehensive guide to bytecode; see the latter sections of this research paper for more information regarding bytecodes and their interpreters):

```
	LDINT32 0 1 # load the 32-bit integer 1 into register 0
	LDINT32 1 1 # load the 32-bit integer 1 into register 1
	LDINT32 2 2 # load the 32-bit integer 2 into register 2

	MULT 1 2 3 # multiply the values of register 1 and register 2, and store the
output in register 3
	ADD 0 3 4 # add the values of register 0 and 3, and store the output in
register 4
	OUT 4 # return the value of register 4 and stop
```

These instructions can be generated through a simple scan-and-match of the AST. For example, when an *ADD* node is encountered, we can generate its bytecode counterpart with the corresponding parameters. Register management (deciding which register to use for each instruction) can be implemented with a single counter that is incremented on each register usage.

The Slang Programming Language - 4.2

This independent study project was an attempt to design a compiler and interpreter for a statically-typed language. For the aforementioned compiler, I created the Slang Programming Language (in short, Slang). Slang was designed with the purpose of being easily readable and useable, like Python, yet familiar to programmers from the more traditional C-style languages. To whet our appetite, let's take a look at a snippet of Slang code:

```
	# This is a single line comment.

	###
	This is a multi-line comment.
	See?
	###
```

```
void run(){
      string word = “hello”;
      print(word); # stdout : “hello”

      int number = 256;
      print(number * 2); # stdout : 512

      if (true) {
            print(“hi”);
      } elif (false) {
            print(“bye”);
      }

      Demo struct_object = new Demo(“Hello, World!”); # implicit constructor
      struct_object.print_field(); # stdout : “Hello, World!”
}

struct Demo {
      string field;
} impl Demo {
      void print_field(){
            print(field);
      }
}
```

Although this is a relatively minimalistic example (for a much more detailed Slang specification, read the official documentation), it does show a lot about Slang's syntactical structure and semantic design. Slang is a statically-typed, statically-scoped language that uses a struct-and-impl based programming paradigm to represent custom data structures. Obviously, Slang borrows a lot of its architecture from other programming languages; with the most notable influences coming from C, Rust, Go, and Python. To an experienced C-like language programmer, Slang should feel quite familiar.

The other item that we focused on while designing Slang was eliminating the state. Slang has no global or static variables. Everything communicates through method calls and return values, thus eliminating any sort of shared, mutable state. As described in the interpreters section, this allows us to make some interesting concurrent statement execution optimizations.

Implementation - 4.2.1

For this independent study project, I was tasked with the responsibility of writing the Slang Compiler (SlangC). The goal of this compiler was to convert the human-programmable Slang code into the machine executable Slang Bytecode (SKB). For the parser and tokenizer, I used the ANTLR parser generator and its respective API to design the grammar. The semantic analyser was implemented in Python, using the ANTLR Python 2.7.6 Runtime API as the internal means of data manipulation. A very simple code generator was completed during the time we had; having the capacity to produce simple arithmetic instructions from parse trees.

Designing an Interpreter for a Statically-Typed Language - 5

One can parse a language and even emit bytecode for it; but, without an interpreter, it is simply an elegant way to represent computation to be performed by humans. An interpreter is what allows computation to be efficient - it is what allows a language to be run by a computer, without the intervention of a human.

Background - 5.1

Paradigms in Bytecode - 5.1.1

In order to design SKB (the bytecode of Slang), it is important to look at other bytecodes for inspiration. As references, I used x86, Lua bytecode, MSIL, and Java bytecode, allowing for a good representation of the two main implementation paradigms for bytecodes - stack-based implementation, used by MSIL and Java bytecode; and register-based implementation, used by Lua bytecode and x86. In a stack-based bytecode, we have what is known as an *operand stack*, upon which we push values, and then operate on the top of it. To illustrate this, we can look at a snippet from the programming language Forth.

```
1 2 + 3 * 9 +
```

Each token (a number, a plus sign, or a multiplication sign, in this case) performs an operation on the stack. A number pushed that number onto the stack; an operator (plus sign or multiplication sign) pops the top two elements off the top of the stack, performs its operation on them, and the pushed that onto the top of the stack. The execution order is left to right. Let's look at the execution of the aforementioned snippet of Forth code, considering the stack as we go:

```
Token  |  Stack
    1  |  1
    2  |  2 1
    +  |  3
    3  |  3 3
    *  |  9
    9  |  9 9
    +  |  18
```

The leftmost part of the stack is the top. Note the notation used for arithmetic in Forth may also be known as *reverse-Polish notation*.

In a register-based bytecode, we use registers where one would traditionally use variables. Registers are usually, although not always, numbered. In x86, there may be three or four ways to refer to the same register. As an example, the register ax is the low sixteen bits of the register eax, which is itself the low 32 bits of the 64-bit register rax, which is also referred to as r0x in AMD-land.

Due to the vast array of academic literature for register-based languages, compared to the relatively limited literature for stack-based languages, and the greater simplicity (which I'll get to later), I chose to go with a register-based bytecode, although there exist many successful languages, such as Java, Python, and .NET, that use a stack-based bytecode.

Interpreters, JITs, Tree-Walkers, and Ahead-Of-Time Compilers - 5.1.2

The traditional way that languages were made was to compile the source code to the native instruction language of the CPU. This is the approach taken by assembly languages, and later by higher-level languages, such as B and C. This is the use-case that most of the traditional literature is focused on, and as a result, it is considered the most well-studied way of implementing a language. This technique is known as an *ahead-of-time (AOT) compiler*.

If we do not wish to run code directly on the CPU, it is obvious that we must use an interpreter. The most obvious way to do this is simply to interpret the abstract syntax tree directly, without any intermediate steps. This is the approach taken by many of my earlier projects from my spare time, as well as some more mainstream languages such as Perl 5 (but not Perl 6), and it is known as a *tree-walker*.

Perl has a reasonable argument for this; during the parsing phase, it is possible that the interpreter may begin to execute code in order to decide how to parse a snippet of code; similarly, at runtime, the interpreter may re-invoke the parser or alter the abstract syntax tree live. Any more complex representation would simply add overhead to these operations.

Although tree-walkers can be very simple and easy to implement, their performance leaves much to be desired. As a result, more modern interpreters, such as CPython, the Lua interpreter, and early JavaScript engines, will compile the AST to a bytecode, and then interpret the bytecode. This brought dramatic performance improvements, although it made the codebases larger and more complex. This technique goes by many names, although the most common one is a *bytecode interpreter*.

However, bytecode interpreters were still much slower than even simple ahead-of-time compilers. This prompted the creation of Just-In-Time compilers, or JITs. These would compile the bytecode, or sometimes the AST, down to the native instruction set of the CPU. Simple JITs can achieve high performance, often approaching that of AOT compilers.

But JITs gain a valuable insight into a program. There are many theorems about a program that are undecidable, or extremely difficult to prove, but can easily be seen to be true in practice, such as types that variables will take, or bounds that variables will fall in.

I chose to implement a bytecode interpreter, as a JIT can often take multiple people many months to develop; I simply didn't have the time.

Slang Bytecode - 5.2

Slang bytecode (SKB for short, extra history points if you can guess why) is a register-based, statically-typed bytecode that is intended for the output of the Slang compiler. It requires all executable code to be within a method. Like MSIL, it is actually textual, and can be written and read by humans. A complete description can be found in Appendix A.

Garbage Collection - 5.3

Modern programmers often think very little about the lower-level aspects of the computer itself; the language handles that for the programmer, freeing their mind for more important considerations, such as the architecture of the program.

One such low-level detail is memory management. Let us consider the following snippet of Java code.

```
byte[] bytes = new byte[256];
for(int i = 0; i < bytes.length; i++) {
    bytes[i] = (byte) i * 13;
}
```

What we are doing is first creating an array of 256 bytes, and then looping through it, assigning a value to each byte. (Incidentally, as 13 is coprime to 256, this will be a permutation of the bytes 0-255). But, where is that 256-byte array coming from? Let us consider the following equivalent snippet of C code.

```
unsigned char* b = malloc(256);
for(int i = 0; i < 256; i++) {
      *(b + i) = (unsigned char) i * 13;
}
free(b);
```

We can now realize exactly what is going on: we are allocating a pointer (essentially, an integer which contains the memory address of the variable being point to), which is at the start of a 256-byte swath of memory. We are then looping through that swath of memory and assigning to it, freeing it immediately afterwards.

This memory is itself taken from a pool of free segments of memory. When it is allocated, it is marked as being used, and then handed to the application, which then writes to it, and then tells the memory manager (or *malloc function*) that the memory may be used the next time it needs to hand a swath of memory to someone else. If we hadn't freed it, we would've ended up with a *memory leak*: we mark memory as used, without ever freeing it, until eventually we run out.

But notice that, in the Java sample, we didn't have to manually tell the memory manager to free the byte array we allocated. *That's because Java does it for us*; when variables go out of scope, they are automatically freed.

But Java supports something even more powerful. Consider the following snippet of Java code.

```
String[] strings = new String[5];
strings[0] = “The first string.”;
strings[0] = “The second string.”;
```

Notice that we never explicitly freed the first string. Does that mean that it will never be freed, resulting in a memory leak? No. When the Java virtual machine decides that it needs more memory, it goes through all the objects currently allocated, and asks whether each one could possibly be used by anyone; whether anyone would notice if it disappears. If the VM determines that it's unreachable, it will tell the memory manager to free it. This is known as *garbage collection*, and it's a common feature of modern languages.

Reference Counting - 5.3.1

Initially, garbage collection seems trivial. For each object *O*, let R_O be the number of references to *O*, anywhere in memory. This is known as *reference counting*, or *refcounting*, and will even work in some simple cases, such as our Java example above. However, consider an implementation of a doubly-linked list.

```
class LinkedListElement {
      LinkedListElement previous;
      LinkedListElement next;
}
```

Consider a linked-list element *E*, at index *i*, and *F*, at index $i + 1$. The next element of *E* will be *F* , and the previous element of *F* will be *E*. Thus $R_E = 1$, and $R_F = 1$, if we ignore the previous element of *E*, and the next element of *F*. This means that the reference count of both *E* and *F* will be quite large; at least 1, and usually 2, 3, or more. Thus, neither *E* nor *F* will ever be garbage-collected, even though they may not be reachable from anywhere.

In other words, neither *E* nor *F* will ever have zero references to them, as each holds a reference to the other, so neither will be garbage collected first.

This is the inherent problem with refcounting. Almost every garbage collector implements at least some form of refcounting, although it is invariably supplemented with some other algorithm to catch the complex cases.

Tracing Garbage Collection and Mark-And-Sweep - 5.3.2

There are many ways to solve the problem of reference cycles. One common solution is to use what is known as a *tracing garbage collector*, which will consider the entire set of currently allocated objects, and decide which ones are reachable.

The most common algorithm for tracing garbage collectors is known as *mark-and-sweep*. In order to decide which objects are unreachable, we start at the "root" object, or whatever the equivalent is in the given language, and add it to the *open set*. We then mark it somehow as being reachable. Next, we look at all the objects we can reach from it, and add them to the open set, and remove the root object from the open set. We then mark them, add all the objects we can reach from them that haven't been marked yet to the open set, and remove the old elements of the open set. This is somewhat similar to Dijkstra's algorithm for graph traversal. Then, any objects that are unreachable are freed.

However, this is relatively inefficient, often pausing execution for the better part of a second. As a result, we often use an algorithm to detect reference cycles in conjunction with refcounting as opposed to mark-and-sweep or the likes. I did not do significant research on cycle-detection algorithms.

Optimization - 5.4

There are many ways to write code that will function correctly, but will nevertheless be unbearably slow. As an example, consider the following C code.

```
#include <stdio.h>

int main() {
    int r = 0;
    for(int j = 0; j < 1000000; j++) {
        int a = 0;
        int x = 3;
        for(int i = 0; i < 1000; i++) {
            a += 6 * i + x * x;
        }
        r = a;
    }
    printf("The number: %d\n", r);
}
```

If one compiles it with GCC with optimizations disabled, it will run in about 2.35 seconds. However, there are many transformations (known in this context as *optimizations*) that could be applied to this code that would improve its performance dramatically, ranging from computing the `x * x` only once to realising that the outside loop needs only be run once to evaluating the entire thing at compile-time.

Standard Optimizations - 5.4.1

The following are examples of standard optimizations. I'm not going to even approach the vast array of techniques outlined in the academic literature; this is simply a taste of what is out there.

Loop-Invariant Code Motion

Loop-Invariant Code Motion comes from the realization that there are certain expressions in loops that are invariant throughout iterations of the loop. Consider the following example, in Java this time.

```
int[][] l; // Initialize this somehow.
long sum = 0;
for(int i = 0; i < l.length; i++) {
    for(int j = 0; j < l[i].length; j++) {
        sum += l[i][j];
    }
}
```

In each iteration of the inner loop, we are adding `l[i][j]` to `sum`. To get `l[i][j]`, we first look up the start of `l`, navigate to index `i`, follow the pointer to the start of `l[i]`, and then navigate to index `j`, and follow that pointer. This is all necessary to be run at some point. However, `l[i]` is invariant to the inner loop; it only changes on each iteration of the outer loop. Thus, we can rewrite the code as the following.

```
int[][] l; // Initialize this somehow.
long sum = 0;
for(int i = 0; i < l.length; i++) {
    int lAtI = l[i];
    for(int j = 0; j < lAtI.length; j++) {
        sum += lAtI[j];
    }
}
```

We can further optimize this by lifting the `.length` at each iteration of each loop up to the containing loop.

```
int[][] l; // Initialize this somehow.
long sum = 0;
int lLength = l.length;
for(int i = 0; i < lLength; i++) {
    int lAtI = l[i];
    int lAtILength = lAtI.length;
    for(int j = 0; j < lAtILength; j++) {
        sum += lAtI[j];
    }
}
```

Loop-Step Optimizations

Consider the following code, which computes the sum of every other element in an array. Note that integer division rounds towards negative infinity.

```
int[] l; // Initialize this somehow
int sum = 0;
for(int i = 0; i < l.length / 2; i++) {
    sum += l[i * 2];
}
```

Beyond the loop-invariant optimizations that could provide a small performance improvement, we can remove the multiplication, and instead increment the loop counter by two each iteration. On architectures such as x86 without a dedicated increment instruction, this could significantly improve performance (ignoring memory bandwidth limitations).

```
int[] l; // Initialize this somehow
int sum = 0;
for(int i = 0; i < l.length; i += 2) {
    sum += l[i];
}
```

This removes one instruction per loop iteration.

Constant Folding and Common Subexpression Elimination

Constant Folding comes from the realization that many constants can be computed at compile-time and substituted at runtime. Consider the following code. It answers the interview question of "Given a list of the numbers, 1 to 100, each given twice, except for one that is given only once, decide which number is given only once."

```
public int getMissing(int[] arr) {
    int sum = 0;
    for(int i = 0; i < arr.length; i++) {
        sum += arr[i];
    }
    return (5050 * 2) - sum;
}
```

The 5050 * 2 is given instead of the obvious result 10100 for semantic reasons. Ideally, this would not incur a performance cost. As a result, many modern compilers will perform the arithmetic at compile-time. We can further extend this to systems with multiple variables.

Another similar, but distinct, optimization is known as Common Subexpression Elimination. Consider the following example.

```
public PointF computePosition(float time) {
      float rotationMultiplier = 4;
      float initialPosition = 5;
      float scale = 1;
      float offsetX = 0;
      float offsetY = 0;
      return new PointF(cos(rotationMultiplier * time + initialPosition),
sin(rotationMultiplier * time + initialPosition));
}
```

Note the presence of `rotationMultiplier * time + initialPosition` twice. This can be reduced into one evaluation.

```
public PointF computePosition(float time) {
        float rotationMultiplier = 4 * Math.PI;
        float initialPosition = 5;
        float scale = 1;
        float offsetX = 0;
        float offsetY = 0;
        float _opt_var1 = rotationMultiplier * time + initialPosition;
        return new PointF(cos(_opt_var1), sin(_opt_var1));
}
```

Dead Code Elimination

Dead Code Elimination aims to eliminate useless computations without affecting legitimate code's view of the world. Consider the following code, remembering that division rounds towards negative infinity.

```
int[] l; // Initialize this somehow.
int b = 2**4 - 1;
int a = 7 - b / 2;
for(int i = 0; i < a * l.length; i++) {
    do_something(l[i]);
}
```

Advanced constant folding would reduce this code to the following.

```
int[] l; // Initialize this somehow.
for(int i = 0; i < 0; i++) {
    do_something(l[i]);
}
```

Dead code elimination realizes that the contents of the loop cannot be reached, and so it is not necessary to even include it. As well, if `l` (the integer array) is only used here, then it would be removed by dead code elimination as well.

Method Inlining

Method Inlining comes from the realization that method calls add a certain amount of overhead. Short method calls would not increase the size of the binary significantly if inlined; however, they could reduce overhead significantly. As an example, if we have a function square, which squares an integer, it may be prudent to inline it.

Tail-Call Optimizations

Consider the following code.

```
int square(int a, int b) {
    return square(new NumberContainer(a), new NumberContainer(b));
}
```

This would be represented as the following SKB.

```
method {
      name square
      params int int
      maxregisters 5
} {
      structctor 0, NumberContainer, 2;
      structctor 1, NumberContainer, 3;
      callret 2, 3, square, 4;
      return 4;
}
```

After the `callret` instruction, no registers other than register 4, the one used as the result of the `callret`, are used; thus, we can actually eliminate the middle method call (i.e. the call to `square(int, int)`) from the stack and thus the chain of returns.

Peephole Optimizations

Peephole Optimizations are small optimizations that replace a given pattern with another pattern known to be faster. As an example, consider the following code, from a stack-based bytecode:

```
ldint 5;
ldint 5;
imul;
```

This would first push a 5 onto the stack, and then another 5 on top of it, then take those two fives, and multiply them together. However, let's say that, on the architecture that this is being compiled for, it is (for whatever reason) significantly faster to duplicate the top of the stack (using the `dup` instruction) than to load an integer onto the top of the stack. Thus, a peephole optimization would be to replace the second `ldint` with a `dup`.

```
ldint 5;
dup;
imul;
```

Optimizations for Dynamic Languages - 5.4.2

As of yet, we've only discussed optimizations that would be performed by a traditional optimizing compiler that operates only at compile-time, and can only see what can be seen statically. However, modern JITs take advantage of information that can glean only at runtime.

SpiderMonkey is the first JavaScript engine ever created; it is now developed by Mozilla. For a long time, it was implemented as a bytecode interpreter; however, in 2008, Mozilla released TraceMonkey, a JIT for JavaScript. In later iterations, they've moved to JaegerMonkey, and then IonMonkey and OdinMonkey.

TraceMonkey is a type of JIT known as a tracing JIT. It will interpret code for a while, and collect information. When it detects hot code, it will use all the information it has available to optimize the function, going as far as reordering clauses in if-statements, making assumptions that are checked before execution begins about types, bounds, and many others. This allows for extremely fast execution of hot code, but also requires a lot of time for optimization. Additionally, only a small fraction of the code is JITed, and switching between the small fraction that's JITed and the vast majority that's interpreted is expensive.

For that reason, in 2010, Mozilla introduced JaegerMonkey, which is much more liberal in the code that it will compile; not only does it compile entire methods at a time, instead of only those small fractions that are used most often, but, due to its speed, it can JIT a large fraction of the code.

JaegerMonkey operates in a different manner from traditional optimizing compilers (and even most modern JITs) in that it simply iterates over the bytecode, reordering and replacing instructions in multiple passes, as opposed to constructing a control-flow graph, optimizing that, and then converting that into the native instruction set of the CPU. This allows incredibly fast compilation times, at the cost of execution times of the compiled code; the compiler was limited to peephole optimizations, constant folding, simple loop-step optimizations, and other optimizations that can be performed without building a complex graph of program execution.

In 2012, Mozilla introduced IonMonkey, which was a more traditional optimizing compiler, supporting many of the traditional optimizations outlined above. In addition to these, however, it supports a very interesting set of optimizations relating to JavaScript's dynamic type system.

Many modern programming languages impose what is known as "static typing discipline" - where a variable can assume one and only one type. Many more languages, however, support the idea that a variable can assume more than one type at different times, and that the type assumed is indeterminate until runtime; this is known as "dynamic typing discipline." It is generally considered easier to write in dynamically-typed languages; however, it becomes hard to maintain as a codebase becomes larger, and it is often slower. To counteract the loss of performance, modern virtual machines for dynamic languages will employ two distinct but related techniques, *type inference* and *inline caching*.

In many cases, it is easy to infer the type of a variable at compile-time; for example, when a variable is assigned a value of a numerical literal. In these cases, we know the only type that a variable may assume, and so we can proceed without any other type checks for that variable. This technique is known as *type inference*.

However, in many cases, it is impossible to infer a type for a variable, such as method parameters, or when a variable may be assigned two different types depending on some conditional or other control flow.

In this case, we can examine the types assumed by the variable in question at runtime. If a variable only ever assumes one type during our observation, we can simply check at the first assignment and in other places in which the type could conceivably change and ensure that it only ever has the type we've assumed. This is known as *monomorphic inline caching*.

However, if a variable has been observed to assume multiple different types, we can upgrade to *polymorphic inline caching*. We perform a similar operating as monomorphic inline caching, except that we emit the code for multiple possible types, instead of just one.

Note inline caching is only useful in the case of a JIT; it's useless in the case of an interpreter. Also, when there are many types that a variable has been observed to assume (the threshold in SpiderMonkey is somewhere around 10), we emit generic code that can deal with any type.

HotSpot is Oracle's Java virtual machine. It employs many of the techniques used in virtual machines for dynamic languages such as JavaScript, including inline caching. Just as SpiderMonkey, it interprets bytecode in unimportant areas, using JIT compilation only in hot spots (hence the name). This allows it to spend more time performing more advanced optimizations, as only a relatively small portion of the codebase is JITed.

Automatic Parallelization and the Advantages of (Partial) Statelessness - 5.4.3

Slang doesn't have static variables. At first glance, this may seem like a limitation. However, in real life, I rarely find myself using static variables except as a temporary substitute before I write a more object-oriented solution; and, moreover, it allows incredible optimizations that would not otherwise be possible.

As an example, consider the following code, in which terrain is generated in a Minecraft-like voxel-based game.

```
public Chunk generateChunk(int x, int y);
public World generateWorld(int xSize, int ySize) {
    Chunk[][] chunks = new Chunk[xSize][];
    for(int i = 0; i < xSize; i++) {
        chunks[i] = new Chunk[ySize];
        for(int j = 0; j < ySize; j++) {
            chunks[i][j] = generateChunk(i, j);
        }
    }
    return new World(chunks);
}
```

The chunk generation algorithm (`generateChunk`) is relatively complex, and takes the better part of a second.

We can see that, as it is, we generate each chunk sequentially, one after the other. However, this is certainly not the optimal arrangement; on a computer with eight cores, as an example, it would be beneficial to use eight threads, to achieve a close to 8x performance improvement.

In most languages, we would need to consider the exact contents of `generateChunk` in order to decide whether it would be safe to run in parallel (as it could affect state, such as static variables, that would violate consistency guarantees and order-of-events for other instances of the function). However, Slang has no concept of static variables. Thus, as the parameters are simply integers, and cannot be modified, we cannot violate any consistency guarantees. Thus, we can easily convert this code to run on eight threads without any extra effort on the part of the programmer.

This lack of static variables has other advantages. As an example, Slang currently has no standard library. Most languages would implement the standard library using a combination of the language itself and C extension modules. We could simply make calls over the network (using local sockets, kind of like X does), and then implement the standard library in any language we want (such as Rust or Java).

Extension modules are not the only advantage of no static variables. With Slang, we can spin up a new process of the Slang interpreter to contain the library, and have multiple main processes use the same library process, with the library itself being none the wiser. This opens up new possibilities for using Slang for lower-level system functions, possibly going as far as writing the vast majority of an operating system in Slang.

Memory Optimizations in x86 - 5.4.4

The rest of the paper will still make sense without this section. Feel free to skip it.

Modern CPUs run at approximately 3 billion cycles per second. In order for a signal to get from the CPU to the RAM chip and back, it must travel through approximately one meter of wire. The speed of light is approximately three hundred million meters per second; thus, we can make this trip approximately three hundred million times per second. But there are 3 billion CPU cycles per second. *It takes approximately 10 CPU cycles just for a signal to the RAM chip to make a round trip, let alone retrieve data and send it back to the CPU*. All told, this can take upwards of 100 CPU cycles.

In response to this, we created the CPU cache. It would sit between the CPU and the memory, and store commonly used values from the memory. There would be three levels of the cache, known as the L1 cache, L2 cache, and L3 cache. The L1 cache is actually in the CPU core; there is one per core, although each one is rather small. The L2 cache is larger, and is shared across all cores. The L3 cache is even larger than that, often reaching 64MB or more. Every memory access would first go out to the cache before going to the RAM chips.

Memory writes would update the L1 cache, L2 cache, L3 cache, and then main memory, in that order. However, consider two threads, each reading and writing to the same segment of memory. It takes time for data to reach the L2 cache, and then time for the L1 cache of each processor to realize that another processor has changed the upstream (L2) view of a segment of memory.

Thus, *we end up with every processor having a different view of memory*; some of their views may be impossible in a serial system. There are many interesting examples we can create with this. Consider the following code.

```
int number1 = 0;
int number2 = 0;
void threadOne() {
    number1++;
    print(number2);
    number2++;
}
void threadTwo() {
    number2++;
    print(number1);
    number1++;
}
```

What you would expect is for both threads to print “1”. However, in some cases, a thread may print “0”, or even sometimes “2”. The reason for “0” is obvious: both threads run at approximately the same time, and it may take many CPU cycles for the memory write to reach the other thread. However, the reason for “2” is not obvious at all. This comes from a very strange property of the x86 memory model that is much beyond the scope of this paper (and that I don’t fully understand).

Caches are useful. However, they are largely for data. For conditionals, caches are still used, but a system known as a *branch predictor* is more prevalent. Consider the following code.

```
uint8_t numbers[QTY]; // This is some random data
int main() {
    printf("Starting...\n");
    int sum = 0;
    for(int j = 0; j < 200; j++) {
        for(int i = 0; i < QTY; i++) {
            if(numbers[i] < 100) {
                sum += numbers[i];
            }
        }
    }
    printf("Number: %d\n", sum);
    return 0;
}
```

(During my experimentation, I used a cryptographically secure pseudorandom number generator based on the XTEA encryption algorithm to avoid patterns in the data). If we time this (make sure that compiler optimizations are disabled), we see that it takes about 1.1 seconds to run. Consider now what happens if we sort the data first.

```
uint8_t numbers[QTY]; // This is some random data
int main() {
    // The following code implements the Radix Sort algorithm.
    uint32_t frequencies[256] = {};
    for(int i = 0; i < 256; i++) frequencies[i] = 0;
    for(int i = 0; i < QTY; i++) frequencies[numbers[i]]++;
    for(int i = 0, freq_index = 0; freq_index < 256; i++) {
        if(frequencies[freq_index] == 0) {
            freq_index++;
        }
        frequencies[freq_index]--;
```

```
            numbers[i] = freq_index;
        }
        printf("Starting...\n");
        int sum = 0;
        for(int j = 0; j < 200; j++) {
            for(int i = 0; i < QTY; i++) {
                if(numbers[i] < 100) {
                    sum += numbers[i];
                }
            }
        }
        printf("Number: %d\n", sum);
        return 0;
    }
```

If we time this code (including the negligible time it takes to sort the data), we see that it takes approximately 0.5 seconds to run. Why is it that sorting the data improves the performance so much?

As we've already established, it can take a long time for data to come back from memory; this is time we could spend executing. We already have caches, but those still aren't instant, and in many cases (such as this one), they aren't even particularly useful. As we're waiting for data to come back from data, we (the CPU) can attempt to guess what will come back from memory. If we've guessed correctly, we've saved valuable CPU cycles. If we've guessed incorrectly, we must backtrack back to where we made the assumption.

When the data is unsorted, we rarely guess correctly, and do so no more often than if we were to guess randomly. However, when we sort the data, we guess correctly nearly 100% of the time (with only a small period when we switch over from numbers less than 100 to number greater than or equal to 100 where we sometimes guess incorrectly), and so we don't have to wait for data to come back from memory, or spend time backtracking.

Armed with the above information, we can now see that an interpreter may never even approach the performance of a JIT. Every instruction must come back from memory. Although the branch predictor will usually be able to predict the instructions themselves (especially in loops), variables must be stored in memory, and cannot be in registers, regardless of the amount of optimization done to the interpreter.

Building a Runtime - 5.4.5

How does one build a bytecode interpreter?

The obvious manner in which one would build a bytecode interpreter is to use a gigantic switch statement; with a case for each opcode. This is relatively slow, as we are checking whether our opcode is equal to *every other opcode*.

The next method that was developed is known as a *table-based interpreter*. Essentially, we look up the opcode in a table, and that gives us a method pointer. We then jump to the location pointed to by the method pointer. The interpreter I wrote is a form of table-based interpreter.

Now, moving into the territory of JITs, we have *inline-threaded interpreters*. Do not be fooled by the name; these are, in fact, JITs. Essentially, we compile the stream of our bytecode to a stream of native x86 by concatenating the x86 implementations of our bytecodes and appending a *ret* instruction. We then jump to the beginning of the buffer we've created. The implementations will be run in order, and then we will hit the *ret* instruction, and return control to where we began. This approach is used by HotSpot on some architectures, and by old versions of SpiderMonkey.

State-of-the-art interpreters will today use a proper compiler; they will simply compile the entire program into an x86 binary, and execute it. There exist tools such as LLVM that will help with this process.

Conclusion - 6

Compilers and interpreters are an integral part of any kind of technological research and development today. Rarely is there ever a digital product in the 21st century that does not utilize a single bit of programmed software. Whether it be a laptop or phone; a microwave or digital clock; at the lowest-level of the software, they can all be drilled down to code written by humans. As programmable software advances, our language recognition applications must improve as well - understanding how the various elements in programming languages are integrated and implemented is valuable information for any developer, IT specialist, or casual software user.

Appendices - 7

References - 7.1

[1] Aiken, Alex, PhD. "Compilers, with Alex Aiken." YouTube. Stanford University, 2012. Web. 29 May 2016.

[2] Gagnon, Etienne, and Laurie Hendren. "Effective inline-threaded interpretation of Java bytecode using preparation sequences." *Compiler Construction*. Springer Berlin Heidelberg, 2003.

[3] Grune, Dick. *Modern Compiler Design*. Chichester: Wiley, 2000. Print.

[4] Knoop, Jens, Oliver Rüthing, and Bernhard Steffen. *Partial dead code elimination*. Vol. 29. No. 6. ACM, 1994.

[5] Morris, F. Lockwood. "A time-and space-efficient garbage compaction algorithm." *Communications of the ACM* 21.8 (1978): 662-665

Process - 7.2

Problem - 7.2.1

For this project, we were tasked with designing a versatile and unique programming language, and creating a fast runtime environment for the execution of its bytecode. Being able to write a compiler and an interpreter from scratch is often regarded as the being the most advanced challenge within the field of formal language theory, due to the intricate understanding of computer science that is needed to construct these programs. Real-world programming language projects, such as Python, Java, C#, Swift, and Objective-C, as well as parser generator projects such as YACC and ANTLR, often take years of research by a large group of developers to complete (with the notable exception of ANTLR, which, for the longest time, was developed solely by Terence Parr). In a time period of three months, we had to produce a working prototype of our programming language, which, considering the above factors, was a significant challenge for the two of us.

Synopsis - 7.2.2

When we began this project, I (Jonathan) already had an intimate understanding of many of the concepts used in compiler design. I had previously read half of *Modern Compiler Design*[3], and I also designed several esoteric and real programming languages; most of which were based off of C. The first main hurdle we had to overcome was to develop a high-level example of the feel and touch of the programming language we were to implement. As seen in the journal, we have several entries detailing the process in which we fine tuned certain aspects of Slang. Following this, I began implementing a Java compiler. Unfortunately, this project turned out to be much too time consuming; thus, I decided to build the parser using ANTLR to save time. After weeks of development, I progressed to a point where the parser was complete and I was able to work on building a framework for semantic analysis and, ultimately, code generation. During the time available for completing this independent study (barring the time we had set aside for writing the paper), I had completed a full framework for semantic analysis, and was able to create working prototype of a portion of the code generator.

I (Ethan) spent the first few days researching; I researched topics such as garbage collection, optimization, and parsing. For the next few days, I was working on an interpreter written in Java, in order to better understand the concepts underlying interpreter design without caring as much about performance. After that, I was writing an interpreter written in Vala, a programming language that, kind of like C++, just compiles down to C, and attempting to squeeze every drop of performance out of it that I could without actually writing a JIT (although I did consider writing a JIT towards the beginning).

Materials - 7.2.3

All of our research and programming were fuelled solely by many hours of hard work, two machines, and a trusty text editor.

Summary - 7.2.4

Through the practical application of the theoretical concepts used to define compilers and interpreters, we learned how to use various softwares and technologies to implement our own programming language. For language recognition, we were given the opportunity to explore the ANTLR parser generator API, and its respective Python runtime library. We also made in-depth inquiries into the algorithms in linguistic theory, and how parsers use them to process machine languages. While designing our interpreter, we learned a lot about code optimizations and their implementations. With this knowledge in hand, we were able to successfully construct a full compiler and interpreter for a new programming language that we designed - Slang.