New College of Florida
Spring 2023

Redex

A proof of concept scripting language in Rust.

Ender Fluegge

Supervisor: Matthew Lepinski

Undergraduate thesis in programming language design development

Major: Computer Science

New College of Florida

Acknowledgements

I wouldn't have been able to do this without the continued support of my professors, who provided me with the tools, experience, and time needed to make this thesis happen.

New College of Florida

Ender Fluegge

Ender Fluegge

Abstract

The implementation of a functional programming language from a drafted syntax is an ambitious undertaking. The average programming language functions with a series of interconnected components, all working sequentially to be able to execute a program. Most scripting languages exist in three major components; A lexer to convert characters into tokens that can be recognized, a parser that can derive meaning and associations from the provided tokens, and a runtime to execute the associations in the form of an AST (Abstract Syntax Tree) generated by the parser. The language I created, Redex, tackles programming language development with a syntax that's heavily influenced by Rust & JavaScript via their variable declarations keywords, and functionality. Lua, a very popular embedded scripting language offers permissive error handling and a lightweight execution environment, perfect for what I desired for Redex to have. Its low-level compiler adheres to modern conventions, though it strays away a bit with its regex-based tokenization implementation to capture strings and variable literals and handle escape sequences in the tokenization step. Redex was developed to be fast to type, with a long term goal of being able to handle HTTP related traffic like acting as a webserver or sending many concurrent HTTP requests. It aims to keep a lightweight standard library, with just the core functions for handling standard input/output and communicating with the outside world. Redex offers features from both Rust and Lua, with an intuitive and natural to write syntax.

Keywords – Undergraduate Thesis, Computer Science, Rust

Contents

List of Figures

1 Introduction

The vast majority of our digital world, from web applications to machine learning, is powered by different programming languages, each tailored to a design preference or specific use case(s). As we seek to improve our digital interactions, we must continuously adapt and innovate these languages to meet our ever-evolving needs.

In line with this necessity, my senior project is centered around the development of a new interpreted scripting language which I have named Redex. This language aims to unite features and design choices from two well know languages with distinctive advantages in their respective fields. This project seeks to bridge the gap between system programming, where Rust excels, and lightweight embedded scripting, where Lua shines, offering a seamless experience for developers.

Rust, a language primarily known for its performance and memory safety, has gained popularity in system programming due to its ability to manage memory without a garbage collector, as well as has an extremely robust borrow system to achieve it's safety. It provides low-level power akin to languages like C++, but with a more approachable and user friendly design. On the other hand, Lua is a lightweight scripting language widely used for configuration, scripting, and embedding into other applications. Lua is especially noted for its simplicity, portability, and versatility.

Redex is designed to combine the strengths of Rust and Lua, offering an efficient and powerful language that is both safe and easy to use. It's runtime will exist solely inside Rust, while incorporating Lua's simplicity and ease of embeddability, making it an ideal language for embedding and scripting.

The project involves designing and implementing the core of the Redex language, developing a parser, lexer, and interpreter, and designing a standard library. The outcome is a versatile scripting language that is efficient and straightforward.

The following sections will delve deeper into the specifics of the Redex language, its design philosophy, and the steps taken in its development. In this paper I will weigh pros and cons of different implementation choices, and elaborate on why I chose the ones I did. My objective was not to create the perfect programming language, rather to delve into my exploratory implementations.

1.1 Glossary

An overview and description of different technologies and concepts utilized throughout the paper, from abstract concepts to full technologies that don't relate enough for an in-depth explanation elsewhere but are still a building block for the programming language.

1.1.1 Regex

Regular Expressions (commonly called Regex for short) represent a standard for matching patterns in provided text. As a whole, Regex is a collection of characters that when linked together allow for precise extraction of specific content from a much broader set of samples. Most programming languages ship natively with or have external user support for extracting text based on the Regex standard. In Rust, it's provided second-hand via a package creatively named 'regex'. Regex often consists of multiple selectors (a list or singular character to look for) with constant characters in-between. An example of a regular expression would be something like this;

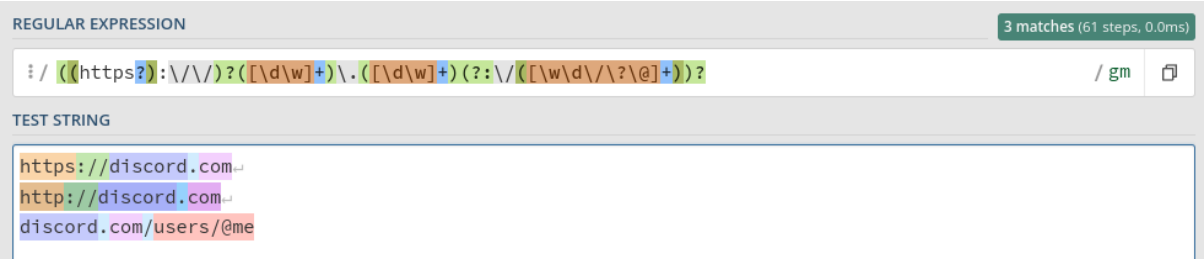

Figure 1.1: A URL regex on regex101

The example above is a regex built to capture information inside of a URL. It consists of three capture groups, the first optional to capture 'http' then an optional 's' at the end. The next capture group gets the domain, the one after gets the TLD or domain extension, and the final one captures the remaining URL path or query parameters. Regular expressions like these can be adapted to all sorts of uses, from machine learning data collection to securing user input and verifying data integrity.

1.1.2 JSON

JSON (**J**ava**S**cript **O**bject **N**otation) is a lightweight data-interchange format that serves to be both extremely fast to serialize deserialize by applications while remaining easy to read and understand for humans. JSON is more or less an alternative to another data-storage format such as XML. It's ideal for representing tree structures, supporting Arrays, Objects, Strings, & Numbers. Within this codebase it's utilized for debugging and saving AST / Parser States with the libraries 'serde' (to handle serialization and deserialization) and 'serde_json' to grant rust structures support for JSON conversion.

2 Compilers, Transpilers, and Interpreters

2.1 Traditional Compilers

In a gruesome over-simplification, Compilers are programs that are fed information in some human legible medium (A programming language such as C++ or Rust for example), and emit machine instructions that correlate to an identical or similar logic as the code provided. The process of compilation usually consists of three major steps (or more depending on the implementation) steps; These are Lexical Analysis, Parsing, and Code-Generation. It's largely accepted that while transpilers are subsets of Compilers, interpreters are not as they don't emit another form of code, but for the sake of convenience and personal bias they will all be included here.

As an example, we'll look at the java compiler:

1. Type Checking Phase: During the type checking phase, the Java Compiler proof reads the code and insures that all variables and expressions adhere to the guidelines defined by the language and version. It verifies that each variable assigned values of compatible types as what they're declared as, and that operations against them make sense according to said types.

2. Symbol Generation Phase: During the symbol generation phase, the compiler checks and validates references to variables, classes, and functions within the code base, making sure they're accessible within the scope in which they're referenced.

3. Byte-code generation phase; This is the last stage before execution the Java compiler takes. It transforms the code into a form of intermediary machine code called

"bytecode", which is then passed forward to the java runtime to be executed. This abstraction on-top of machine code allows java code to be executed on many different platforms with minimal changes or compromises to the code base.

The steps compilers take change vastly depending on what language you examine, such as many of the C++ compilers all tackling compilation in various ways. The steps listed above while don't strictly adhere the Lexing, Parsing, and Code-Generation each step of the Java compiler implements one of those in some capacity.

2.2 Transpilers

Unlike Compilers, Transpilers forego the last step of machine code / byte code generation to instead focus on generating another form of source code in another language or format. This produced code is then passed into an interpreter or compiler to be executed or converted into VM (Virtual Machine) code or native code depending on the language. A popular example of a programming language that does this would be TypeScript or CoffeeScript. Both of these languages provide an alternative syntax to JavaScript, with TypeScript adding strict type signatures which normally don't exist. When executed though, the source code of either of these languages are emitted into JavaScript, then executed by a valid JS runtime such as NodeJS, Bun, or the Browser.

```typescript
class MyClass {
    private prop1: string | undefined;
    private prop2: string | undefined;
    myMethod(): string {
        return "Henlo";
    }
}
```

(a) Typescript

```javascript
var MyClass = /** @class */ (function () {
    function MyClass() {
    }
    MyClass.prototype.myMethod = function () {
        return "Henlo";
    };
    return MyClass;
}());
```

(b) Compiled JS

Figure 2.1: TS transpiled into ES5 JS

2.3 Interpreters

Interpreters / Interpreted Languages don't fall under the general definition of compilers, and with good reason– Interpreters are simply programs that execute code without an intermediary conversion. While they still follow several steps that many compilers do, like pre-processing, lexing, parsing, and sometimes type checks, they never emit to machine code or byte code. Instead, the runtime determines how to handle and execute code, often as it's read line by line. Another great example of a language that does this is once again, JavaScript.

If we have some example code like so;

```
let x = 10;

let y = 20;

let z = x + y;

console.log(z);
```

In the above example an interpreter will execute this code line by line, without any compilation step. It will evaluate and execute the instructions directly:

1. Declare a variable y and assign the value 20 to it.
2. Declare a variable x and assign the value 10 to it.
3. Declare a variable z and assign the value of the expression x + y to it.
4. Call the console.log() function with the value of z as an argument, which in this case will print 30 to the console.

It's important to note the boundary between compilers and interpreters though. Some

languages can both be compiled and interpreted, a common implementation of this would be something similar to V8 JavaScript's JIT (Just In Time) compilation, where certain code is compiled to machine code at runtime, while other parts are still interpreted. This type of hybrid approach tries to improve the speed of performance-critical components while still holding onto the flexibility of an interpreted / scripting.

3 Lexing / Parsing

3.1 Lexing

Lexing, also referred to as tokenization or lexical analysis, serves as the initial stage in the compiling/interpreting process for the majority of programming languages. This phase is the first step in creating machine-readable and interpretable characters and tokens, where tokens are the programmatic representation of each character or in some cases, series of characters. They're the building blocks of programming languages, containing symbols like reserved words (if, else, function, ect), literals (string w/ contents, int w/ contents, and booleans), as well as operators and punctuation marks for mathematical expressions. Lexing aims to act as an intermediary to allow the parser in the next step to have an easier time forming meaningful relationships from the raw content. For clarity, I'm using the term lexing and tokenization interchangably in my description. A Token is often an enum field or class structure, and tokenization refers to generating those tokens from an input, whereas lexing implies generation of tokens with associated data (like position, contents, ect.)

As for how to go about writing a lexer for a given programming language, there are many options, each with their own pros and cons, and with varying speed and memory costs. For Redex's implementation, three were on the table for consideration.

- **ML/AI based Tokenization:** By the time I stumbled upon ML/AI tokenization

as a possability, the regex-based tokenizer already performed flawless, while the logic could be added it was unnecessary. Machine Learning based tokenizers utilize neural networks or decision trees that are trained on the provided input. A tokenizer that utilizes a model or neural network could, in theory, produce far more structured tokens than other Map based or Regex based implementations, possibly skipping or replacing large portions of how the parser processes tokens as well. The drawback of this approach would be the time required to curate and train a dataset for the model, as well as the difficulty of implementing a model that's specifically designed for tokenization for programming rather than natural language.

- **Table/Map based Tokenization:** Instead of using regular expressions, Table tokenizers use lookup tables to map input text to tokens. These tables can both be generated manually or via open sourced tools, and while lookup tables offer one of the fastest implementations of tokenization, they're more memory heavy than other approaches due to having the entire table loaded which in a complex language could be pretty large.

- **Regex-Based Tokenization:** Regex based tokenization is what I selected to implement for Redex, in part due to my familiarity with Regex & the odd sense of enjoyment I get writing regular expressions. The two main types of regex based tokenizers are either extraction based or sequential based. Extraction based regex tokenizers will iterate through a list of regex expressions and compose a list of tokens where they occur, then sort them into the proper order in which they occurred. Because each regular expression is a global match, each regex will only need to be executed once against the entire text, saving a little bit of time per match if the regex is pre-compiled in both examples, and a lot more if it's compiled ahead of

time.

The other approach and the one I decided to go with for convenience, was iterating through a list of Regex expressions, with each expression scanning for an occurrence of it's symbol at the start of the code using the '^' token. Because multiple inputs may match the same symbols, the order of the regex list is extremely important to make sure there's no conflicting matches throughout the code. For example, if you were parsing for identifiers before you were parsing for keywords, the word 'function' may be captured as an identifier / variable name rather than the reserved keyword it should be.

The Rust programming language has an extremely powerful inheritance system, allowing structures and enums to derive traits, helper functions, and other useful associations. When working with enums, a library called strum (https://docs.rs/strum/latest/strum/). This allows for enum fields to have associated properties such as short values, descriptions, and numeric representations. For Redex, this allowed the entire association between token and regex to be defined from a single enum, utilizing Strum's 'IntoEnumIterator' feature to iterate through it.

```rust
#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash, EnumProperty, EnumIter, Serialize, Deserialize)]
pub enum TokenType {
    // Keywords
    #[strum(props(regex = "^let"))]
    Let,
    #[strum(props(regex = "^fn"))]
    Fn,
    #[strum(props(regex = "^if"))]
    If,
    #[strum(props(regex = "^else"))]
    Else,
    #[strum(props(regex = "^return"))]
    Return,
    #[strum(props(regex = r#"(?:^'((?:\\\\|\\'|[^'])*)')|(?:^"((?:\\\\|\\"|[^"])*)")"#))]
    StringLiteral,
    // Literals
    #[strum(props(regex = r#"^\d+(?:.\d+)?"#))]
    IntegerLiteral,
    #[strum(props(regex = r#"^(true|false)"#))]
    BooleanLiteral,
    // Operators
    #[strum(props(regex = r#"^="#))]
```

Figure 3.1: The main Token Enum

With Redex's lexer, given the input 'print("Hello World!")', the parser begins by going through the first regex.

- print: Identifier (28th in the enum)
- (: LParen (16th in the enum)
- "Hello World!": StringLiteral (6th in the enum)
-): RParen (15th in the enum)
- : EOL (Automatically inserted) (N/A)

Parsing that input takes 4 loops across the TokenType enum, executing a total of 65 regular expressions to process the inputted string. Certainly not the most optimal approach to lexing, but still a very adaptable and easy to understand approach to extracting tokens

from text. The match loop takes a few other things into account as well– after iterating through to find out the corresponding TokenType, it constructs an instance of the Token structure, which contains token_type, literal, start, end, and size. All this information provides the parser with what it needs to perform groupings or location-specific error reporting.

3.2 Parsing

The next step following tokenization is Parsing, also known as syntax analysis. This stage exists to take in the sequential tokens provided by the lexer and turn them into dimensional structures, such as Parse Trees or Abstract Syntax Trees, objects that represent the execution flow of the program. This step insures that both the code is free of syntax errors, and that the code adheres to the desired grammatical structure of the language. Over all, this is the most crucial stage and one of the hardest to implement, especially if you're not directly compiling to machine code.

After lexing is completed, the compiler has a stream of sequential tokens that represent each individual element of the original code, including all keywords, literals, operators, and other tokens. Some implementations of compilers opt to trim out white space, such as tabs, spaces, indentations, or new lines, while others continue to feed them through the program, though it boils down to preference over anything else. Even if white space is removed, if each token still has a positional parameter there isn't any loss in ability to syntax check or report accurate errors as the white space can simply be assumed if need be.

Redex's syntax is based loosely off of Rust and Lua, while it's AST generation resembles JavaScript, as it's interpreted nature made it a great example for this language as well.

Listing 1: JavaScript Source Code

```
function myFunction() {
    console.log("Hello!")
}
```

The JavaScript code above directly generates this AST before being executed by the runtime. The necessary code to keep track of the variable's scope, reference its content, and maintain other useful properties is available from here. Similarly to the Lexer, there

Figure 3.2: The generated AST

are many varying approaches to Parsing as well, each naturally with their own pros and

cons, catered to different types of languages and syntax and varying speed / memory constraints.

- **Top-Down Parsing / Recursive Descent:** Top down parsing works by starting at a root node of the parse tree (either automatically created and appended to, or derived from the first token from the token stream) and attempts to match every input token by slowly expanding it's window of input to encompass more tokens until it discovers a sequence that represents valid grammar for the language. The 'recursive' component of this method comes into play when it goes over the grammar a layer deeper, ignoring the starting sequence and ending sequence to instead isolate out the body of the discovered block or sequence. This process is repeated over and over until only a literal is yielded, or there is simply nothing left to scan, and that's when parsing is complete. While Recursive Descent Parsers are relatively easy to implement, they lack ability to handle certain grammars like left-recursion, where they would struggle to make a connection without intermediary logic.

- **Bottom-Up Parsing:** Bottom up parsing works by discovering the lowest level of the tree from the provided tokens, and instead works it's way up to form the complete tree using the grammar rules of the language. Where Top-Down parsing attempts to discover the left most connection, Bottom-Up takes an input and attempts to reduce it to the starting symbol or the smallest form it can. While able to take into account a lot broader grammars, this type of parser is far harder to implement and far beyond my comfort zone.

- **Others:** There are many other approaches to parsing such as Chart/Dynamic parsing that's designed to handle varying grammars, Packrat parsing which is a

> "technique for implementing parsers in a lazy functional programming language", and many others. When it comes to parsing, there isn't a strict right or wrong, rather it simply depends on your use case or comfortability with the logic required in implementation.

The Redex AST as mentioned before follows closely to what a JavaScript AST looks like, minus several unique identifiers for many of JavaScript's verbose features. There exists a primary enum containing all Expressions, with a property for Literals, Assignments, Math, Groupings, Blocks, Program, Return, Token (For passthrough), Conditional, Function, and Call. Each property has an associated structure that contains all relevant properties. For example, an assignment expression contains Meta (start/stop of the expression and length of all tokens inside the expression), as well as an identifier that's being assigned to and an expression that represents the value. A conditional expression is similar, containing the Meta property, a field for the condition, a field for the expression if true, and then an optional field if there is an else statement within the block.

Because Redex is a scripting language, many expressions are simply passed around from structure to structure without being fully processed– In the case of an if statement, the executed expressions aren't ever fully evaluated until runtime of that block of code. While this methodology does provide a decent overhead due to the nature of continuous parsing, it allows for a very fluid implementation of the parser.

Redex uses a rudimentary Recursive Descent Parser to form an AST from a provided stream of tokens. It was built mostly through trial and error and experimentation to learn how to make a language that reflected how I think rather than take the best possible approaches at every turn. The Parser operates using a main loop that iterates over several

steps in order to derive the best next expression. With the input being a vector of Tokens, the steps are as follows:

- Pop the next token, return None if end of input.
- Determine it's kind through the token_type field on the token.
- Run a Match/Switch statement on the token type. Here the grammar is defined for all of the language's notation.
- If TokenType is a literal of any kind, resolve it's type (boolean, string, or integer) and return it as a LiteralExpression that stores the value associated.
- If TokenType is a math character of any kind, return an empty MathExpression that has no left or right attribute. (A lookbehind look ahead will define these properties in the next iteration)
- If TokenType is a Left Brace, perform a lookahead to capture all tokens between this left brace and the next balanced right brace. Return an expression group with the captured tokens as children.
- If TokenType is a Left Parenthesis, Perform a lookahead to capture everything between it and it's next balanced right parenthesis, then drain all commas in case this is a function declaration or function invocation. Perform a check to see if they're all identifiers, if so, this is a group expression and it gets returned as such. If not, this is a ExecutionExpression, which causes the children to be automatically parsed. This is implemented to both handle function related tokens as well as math operations.
- If TokenType is a 'Let' literal, skip the next token (Must be equal sign, else syntax

error.) Then the next token is captured, and returned as an AssignmentExpression with the identifier and the expression set.

- If TokenType is a 'Return' literal, capture the next token (if it exists) and yield a ReturnExpression containing the captured value.
- If TokenType is an 'Fn' expression, assert that the next expression is a GroupExpression, then assert that the expression after that is a ScopeExpression. From there, an AssignmentExpression is returned that binds the identifier (function name) to the scope with the arguments included.
- If TokenType is an 'If' expression, assert that the next expression is a condition, and the one after is a block, then perform a lookahead to see if there is another block which would be our 'Else' expression. From here, a ConditionalExpression is returned.
- If TokenType is white space, continue.
- For **any other token type** we can assume it will be an Identifer, as that is the only unchecked property. A lookahead is performed to see if there is a GroupExpression after it, if so it must be a function and a CallExpression is returned. If not, the token is passed through as a TokenExpression (Expression::Token)

This parser heavily relies on recursive calls to the .next() function, which runs this loop and pulls the next token. The next function 'consumes' all characters it processes in an unrecoverable manner, so if lookaheads are needed for checks, 'peek_next' is called. This function returns a clone of the Parser at the current state of input. From here, as many tokens can be consumed as necessary without impacting the original parser feed. For instances that require a scope / block to be collected (Curly Braces or Parenthesis),

the Parser contains a helper function called 'parse_vec', which is able to take a vector of expressions or tokens and construct the final Expression from them using the same recursion the parser utilizes. Alternatively, 'parse_all' can be used to extract the final expression from a parser created by the peek function.

This approach and dependency on lookaheads makes prototyping and conceptualizing additions to the language easy, while making implementations or changes extremely difficult due to unexpected behaviour of the .next() function on the parser. Of all Redex's code, most issues and problems arise from this recursive descent parsing approach, both as a result of my inexperience in implementing something like this prior as well as Rust's static nature. Assumptions that I know to be true require shortcuts or alternative approaches to actualize due to Rust's advanced memory management practices and lifetimes. Cyclic references are very difficult and often impossible to manage within Rust, so in the parser's implementation there are many copies and clones of data. To cut down on this where I could, I utilized the Box<> datatype Rust offers to manage pointers to Heap Allocated data rather than working with the direct pointer or data itself. Without boxes, a recursive data structure such as Expression couldn't exist, as it's memory size wouldn't be able to be calculated at compile time.

Once a block of Redex code is compiled, the entire tree exists under the root node 'Program', which contains a start and stop point for the program. From here, the tree is ready to be either exported to JSON using serde serde_json's derived methods, or simply executed by the runtime which works very close alongside the Parser.

3.3 Runtime

The runtime is the one part of Redex's implementation that wasn't researched or based upon an existing technology. Not for a lack of understanding, rather it's something I wanted to approach blind, and figure out how to piece components together on my own through experimentation.

The most essential components of the runtime for a bare-bones interpreted language without any unnecessary features to be able to evaluate code are: variable storage, single or multiple conditional flow, and functions or some block-like code abstraction. While each one can be added upon to be far more complex and intuitive, for a 'working' programming language, those are the primary requirements. The runtime exists as a thread-safe Rust structure, with a property for the scope, a count of all instructions (expression evaluations) executed, the start time of the first execution, and a hashmap containing references to the standard functions for Rust Redex interpolation. Scope is it's own structure, acting as a simple abstraction on top of a stored hashmap, storing associations between the identifier name (string) and a 'VariableStorange' instance, which is capable of storing an Integer, String, Boolean, Function, or a recursive scope.

With a simple variable storage comes the next step, while not completely necessary, was a nice feature to try and implement– dynamic casting. Every primitive type within Redex has it's own associated rust file for the structure, implementing a trait called 'StdConversions'. StdConversions offers a way for one primitive to be converted into another, with it's own function defined for each conversion type. For example, the 'to_integer' conversion for the primitive CompoundString invokes the rust f64::parse method on the string, converting it into an integer or throws a runtime exception if the conversion is impossible. Likewise,

the boolean primitive's conversions handle conversion from a boolean to integer by simply returning 1 as the universal truthy value, or 0 for falsy, whereas converting to strings simply returns the true/false as a string. Between all the datatype conversions, functions to other values are where things get unique. Converting a function to a string constructs a template string of the format "Function: name <pointer>", where pointer is an index within the hashmap corresponding to the function's location within the runtime, not the stack/heap. Converting the function to a boolean will always return true, as the function will always be defined, and conversion to an integer will attempt to unsafely read the raw Rust pointer as a reference to the callable function.

```
5 implementations
pub trait StdConversions {
    fn to_integer(&self) -> super::Integer;
    fn to_bool(&self) -> super::Bool;
    fn to_compound_string(&self) -> super::CompoundString;
}

3 implementations
pub trait Callable {
    fn call(&self, runtime: &Runtime, parent: super::Scope, args: Vec<super::VariableStorage>) -> Option<super::VariableStorage>;
    fn name(&self) -> String;
}
```

Figure 3.3: Traits belonging to Functions Variables

On top of implementing StdConversions, rust interpolated Redex functions must implement a trait named 'Callable'. This trait declares two functions, one allowing for the function to be called via a .call method that takes in a runtime, a parent scope, and a VariableStorage vector for parameters, returning an Option containing a VariableStorage in the case the function returns a value. It also contains a name function, which simply just returns the name/identifier of the function, as interpolated functions don't share the same declaration and definition within the runtime that internal user-defined functions do.

Because interpolated functions have a differing initialization, binding the standard library functions is done as a method call through the runtime class, named 'link_std'. This function when run, creates a new VariableStorage Function, with a string name, parameters,

```rust
1 implementation
pub struct ReadLn;
impl Callable for ReadLn {
    fn call(&self, _runtime: &Runtime, _parent: Scope, _args: Vec<VariableStorage>) -> Option<VariableStorage> {

        // Hold the process until standard input is received.
        let mut input: String = String::new();
        std::io::stdin().read_line(buf: &mut input).unwrap();

        Some(VariableStorage::String(
            CompoundString::from(input)
        ))
    }

    fn name(&self) -> String {
        "print".to_string()
    }
}
```

Figure 3.4: The Redex PrintLn standard function

null children, and a ptr, representing a hashmap pointer. Pointers are stored as an option property within all function types in Redex, though it's only in interpolated functions that they're given a value. if a 'call' is invoked through Redex code, it pulls the function by alias from the scope, and checks for a ptr. If there's a ptr variable, it executes the associated function via the hashmap, and if not it will execute the function by evaluating the 'children' vector that it's provided. While a better implementation certainly exists, it was the best way I could determine to allow for reusing function code through Rust's strict type checking.

```
========= CORE DUMP =========
Runtime Size: 120 bytes
Instructions: 3
Runtime: 189
--------- SCOPES: ---------
Global Scope:
println: Function(Function { name: "println", parameters: ["value"], children: [], ptr: Some(1) })
readln: Function(Function { name: "readln", parameters: [], children: [], ptr: Some(2) })
print: Function(Function { name: "print", parameters: ["value"], children: [], ptr: Some(0) })
x: String(CompoundString { store: "hello" })
```

Figure 3.5: A simple coredump

4 Syntax & Future Development

4.1 Syntax

When the programming language was first thought up, my intention was to model it with a syntax similar to Rust and Lua, simply due to personal preference and ease of understanding. As I continued working on the lexer and parser, certain constraints such as lookaheads or capture groups started to sway it's syntax to resemble the C-Like syntax over my original concept.

A good example of this is Rust's if statements,

> Rust: if x == "temp" { /* Do Something */ }
>
> Redex: if (x == "temp") { }

In this example, an if statement doesn't require a matching set of parentheses around the conditional. While I wanted to emulate this behavior, the way I handle lookaheads when the character '(' is detected prohibited me, and devising a workaround might cause other issues down the line.

Listing 2: Function Definition

```
1 fn myFunction(param1, param2) {
2     println(param1)
3     println(param2)
4 }
```

While maps can be defined both in language via rust interpolated functions, indexing them still needs to be worked on. If it's desired to update an already declared variable, redefining it with the 'let' keyword will override it.

Listing 3: Variable Declaration

```
let str1 = "doublequote"
let str2 = 'singlequote'
let str3 = `backtick`
// Nested dictionaries are not supported yet
let dictionary = [
    "key1": "stringValue",
    "key2": 8675309,
    "key3": false
]
```

Conditional logic is able to take in any singular expression, including math functions. All types will be casted to a boolean, resolving true or false based on what they are. For example, an empty string "" will resolve to false, while one that has contents will resolve to true. Likewise, 0 is falsy and 1 is truthy.

Listing 4: Conditional Flow

```
if (condition) {
    // . . .
} else { // Optional
    // . . .
}
```

The condition in a while loop will be evaluated before each loop, with the same restrictions as an if statement.

```
while (condition) {
    // . . .
}
```

Due to the nature of how the AST is constructed, all unencapsulated math is greedy to the right-most expression.

```
let addition = 2 + 2
let subtraction = 2 - 2
let division = 2 / 2
let multiplication = 2 * 2
let modulo = 2 % 2

let x = 10 * 9 - 8 + 7 / 6 % 5 // 4.833~
println(x)
```

FizzBuzz is a common programming interview question, and is a basic test of a person's programming skills as well as serves to demonstrate their ability to use flow control, loops, and modulo operators. For a range of integers, print "Fizz" if said integer is divisible by three, and "Buzz" if it's divisible by five. If both, it should print "FizzBuzz", and below is a working implementation of this problem in Redex.

```
12      let i = 0
13      while (true) {
14          let a = i % 3
15          let b = i % 5
16          print(i)
17          if (a == 0) {
18              print("Fizz")
19          }
20          if (b == 0) {
21              print("Buzz")
22          }
23          println("")
24          let i = i + 1
25      }
```

4.1.1 Standard Functions

As well as supporting basic programming logic, Redex comes with a few basic standard methods for interacting with the user and controlling the program's flow. The runtime contains a function called 'bind_std', which when invoked binds each rust interpolated function into the program's runtime, letting them be accessed by all future executing code. That function can be removed if there is no need for standard functions.

1. println(param) – Prints the passed in property to the terminal with a new line.
2. print(param) – Prints the passed in property.
3. sleep(ms) – Halts the program at the position until the time provided elapses.
4. net_get(url) – Performs a network request, and returns a dictionary containing the status and contents.
5. readln() – Reads input from the user, returning what it received.
6. coredump() – Prints all global variables, the number of instructions processed, and the memory taken up by the runtime.

4.2 Future Work & Planned Implementations

4.2.1 Block Scopes

While Redex does employ the bare minimum features for a working programming language, there's still quite a lot under the hood with it's parser and lexer, as well as many runtime features that need to be fleshed out and worked on. The most prominent missing feature is control of scope. When a new variable is defined, or a function is created it will only

exist within the main global scope of the program, even if it's declared within a function or block. For example,

```
26 fn myFunction() {
27     let x = 55;
28 }
29
30 myFunction()
31 print(x) // This will still print x as 55.
```

The primary reason this wasn't addressed in the initial Redex implementation is due to rust's safety. The runtime has ownership of all members, IE all functions, stored variables, ect. In order to pass scopes through functions, it would require a layer of abstraction on top of the runtime, a place for variables and scopes to live so that they can be passed freely to and from methods implemented in Redex code or raw Rust code.

4.2.2 Dictionary functionality

While I didn't plan on getting to implementing dictionaries in this iteration of development, I did lay the groundwork for their use within the code base. As it stands now, dictionaries can be created by both Rust interpolated functions and from raw Redex code, though they don't support indexing which makes them ornamental at best. Lua's implementation of dictionaries (called Tables in Lua) double as a way to store key value pairs, as well as provide support for the array primitive that most programming languages have. This allows you to either loop over it as key values through use if a function called 'pairs()', or loop through it as an indexed array with 'ipairs()'. This is the same type of behaviour I plan on implementing for Redex in the future.

```
32 let myDictArr = ["key1", "key2", "key3"]
33 let myDictKeyStore = ["key1": "value1", "key2": "value2"]
34
35 for (index, (key, value) in myDictKeyStore) {
36     println(index)
37     println(key) // In the case of an array dict, this would
       be null.
38     println(value)
39 }
40
41 // Indexing it as a map by key would look like the following
42 println(myDictKeyStore.key1) // Would throw an error if
   there's no associated key.
```

4.2.3 Error Handling

Both Rust and Lua have their own unique error handling strategy. Rust provides an 'Option' structure that can be returned from functions, which stores two results, one for a success value and one for a failure value. This structure allows for several methods to either propagate the error to a higher call, unwrap it and throw the error, or do conditional logic against the error without even needing to unwrap. Lua on the other hand, has no error handling (though it offers a 'protected call' method to stop error propagation from creashing code). If you're writing libraries in Lua, and something may return an error, usually the function will return a tuple where the first property is an error or undefined if

there isn't one, and the second property is the value.

For Redex, I wanted something in the middle between these two. A structure similar to Rust's Rust that doesn't support propagation but does provide a singular type for encapsulating errors and performing logic against them.

```
43 fn myHttpFunction() {
44     let result = net_get("https://google.com/")
45     if (result.isError()) {
46         println(result.inner())
47     } else {
48         println("error!" + result.error())
49     }
50 }
```

5 Conclusion

The primary aim of this project was for me to create a unique scripting language that adapts features and syntaxs from the system's programming language Rust, with the embeddable and light weight programming language Lua. This objective has not only been met, but implementing it has expanded my perceptions and opened me to new avenues for exploration in the realm of programming language development.

Through the development process, I successfully established the foundation for a language that can serve diverse coding environments efficiently. Redex, even in its early state with it's Rust based runtime shows a lot of speed and potential for more as implemented practices are improved upon. It's early version managed to reflect my goals of performing simple logic flow, variable declaration and support functions bound in rust.

The completion of this project does not mark the end of Redex's journey but rather the beginning. The development of a programming language is a continuous process, with each iteration aimed at refining its functionality and optimizing its performance. This iteration is only my third, with each improving on the fundamentals of the last. With the desired implementation of new features, a redo of the runtime and core parsing logic will be necessary not just to move forward, but to lay a better foundation to build upon in the future.

Redex is more than just a senior project, it's a floor that's taken into account what I've learned with my time at New College that I can build from. This endeavor has provided me with invaluable insights into language design and has reinforced the idea that progress is a continuous and recursive process, a journey rather than a destination. I look forward

to seeing Redex grow and adapt to new features and standards in the ever evolving field and perhaps one day grow into something that will get used publicly.

6 Appendix

6.1 Source Code

All the code is made available through the GitHub repository located at https://github.com/furry/redex , providing a table of contents to jump to the different implementations (Lexer, Parser, and Runtime) within the code as well as instructions to build it manually.

References

[1] R. Nystrom, "Crafting Interpreters," n.d. [Online]. Available: http://craftinginterpreters.com/

[2] M. Liao, "Building a statically typed Python (part 1: Lexing and parsing)," Medium, October 13, 2018. [Online]. Available: https://medium.com/@mikeliao/building-a-statically-typed-python-part-1-lexing-and-parsing-682b3e50d9c6