Department of Mathematics
and Computer Science

University of Southern Denmark

Bachelors Thesis in Computer Science

Eridu - an Alternative Type System for a C-Like Language

Author
Frederik Gram Kortegaard
frkor19@student.sdu.dk

Author
Lasse Usbeck Andresen
lasan19@student.sdu.dk

Supervisor
Kim Skak Larsen
Professor

June 1, 2022

Abstract

English Compilers and programming languages come in many varieties, each of which brings with it its own set of possibilities and limitations. Since the early days of computing, compilers have become increasingly complex, enabling a multitude of different programming paradigms and shaping the way developers work and think. In this thesis we will discuss various aspects of the design and development process of a modern compiler, showcase the Eridu language and the compiler thereto, and discuss the details of its implementation. We will attempt to showcase how a C-like imperative language such as Eridu can be used to create features greater than the sum of its parts, and finally we will present the results of bench marking Eridu against the C and C++ languages.

Danish Compilere og programmeringssprog kommer i mange varianter, og med hver af disse varianter følger der en række af egenskaber og begrænsninger. Siden begyndelsen af både programmering og datalogien som vi kender den, er kompleksiteten af compilere steget hvilket som resultat har åbnet op for et væld af nye programmeringsparadigmer, samt været med til at forme både hvordan udviklere tænker omkring datalogiske problemstillinger, men også hvordan de bliver benyttet i praksis. I løbet af dette project vil vi diskuttere forskellige aspekter indenfor design og udvikling af compilere, samt fremvise programmerings sproget Eridu og dets tilhørende compiler, og diskutere de design og udviklingsmæssige processer som vi har taget stilling til. Vi vil også prøve at vise hvordan C lignende imperative programmeringssprog ligesom Eridu, kan blive brugt til at skabe funktionalitet der er større end summen af dens dele. Til sidst vil vi præsentere resultaterne af sammenligninger mellem Eridu og programmerings sprogene C og C++

Contents

1 Introduction

Abstract data types form the basis of many higher level programming concepts, the specific implementation of these data types each bring their own set of limitations, capabilities and variations. Throughout this thesis we will discuss not only the implications of abstract data types, but also look at how they vary with regards to the design and implementation of a programming language. Many of these abstract data types create synergies, allowing for more complex abstractions and unlocking an array of different programming paradigms. Expanding on the former description of this thesis, we will also showcase examples of how these paradigms can be used in a simpler programming language. Specifically we are going to look into alternative implementations of object-oriented design patterns, functional programming using higher-order functions, and the implications of different types on the programming experience using the Eridu language.

Furthermore we note that the primary goal of this thesis is not only to discuss type systems and data types, but also discuss general techniques for designing and optimizing compilers as well as intermediate- and assembly level representations of a program.

1.1 Goals & Language Philosophy

When we initially began the planning and design of Eridu, we first set a few clear goals for what we wanted to accomplish with our language. Firstly, we wanted to create a C-like imperative programming language, partly as an homage to *C*, but more importantly, we felt that a low-level imperative language gave us the best opportunity to explore a wider spectrum of functionality, as alternatives such as object-oriented and functional languages require a larger quantity of paradigm-specific properties.

We felt that an important part of developing a language, and being passionate about it, was infusing our own ideals and tailoring it to fit the style of programming we enjoy the most. As a means to personalize our language, we formalized a set of features and stylistic choices that we during our education and personal projects came to favor. Specifically we wanted to promote the use of a verbose and explicit programming style, facilitating a more self-documenting style of development, and reducing the ambiguity of software written in our language. This gave rise to a type-system which enforced strong- and static typing, rejecting concepts such as type inference and dynamic typing. While this decision resulted in a relatively safe core, we in no way wanted to obstruct the power of the language, as its low-level nature implied the ability to work directly with memory by means of pointers.

However, as will be discussed later in this thesis, we did stray away from these ideals with regards to some of the more obscure features of our language, giving developers the freedom to break these constraints, and assume full responsibility of the correctness of a program.

2 The Eridu Language

The first matter of this thesis will be giving the reader a description of the features present in Eridu, as well an an overview of how to write valid Eridu code. In the following sections, we will describe the full scope of features our language provides with relevant code examples to accompany most descriptions. Before delving into more complex features and data types, we first have to briefly discuss the basics of the language.

2.1 Primitives

In Eridu, four primitive data types can be found. These types are - *char, int, float, and str*, whom all share relatively similar operations and descriptions. We chose these four primitives for Eridu, as combinations of these are all that is required to accomplish any given programming tasks. While *str* is not strictly required, we felt that it lowered the barrier to entry for using the language, abstracting away the notion of pointers which in languages such as C, are essential for string manipulation.

Both declarations and assignments in Eridu are essentially identical to that of most imperative programming languages. The following is a simple example of both variable declarations and assignments thereto:

```
char a = 'b';

int b = 5;

float c = 6.0;

str d = "Hello_World";
```

Eridu also supports single line comments, which do not affect the way the code is compiled or executed. We felt that having only single- and not multi-line comments would be acceptable, as the former can trivially emulate the latter.

```
float a = 5.5; // Declares a float with the symbol a
int b  = 5 // Declares an integer
```

2.2 Arithmetic Operators

Eridu provides five arithmetic operators that can be used with both the int and float types.
As with other C-like languages, in Eridu arithmetic operators function as per the following example with the + operator, which takes two expressions of the same type and adds them together.

```
float a = 5.5;
float a_times_two = a + a; // a_times_two = 11.0
```

More formally, for integers, the following binary arithmetic operations are implemented: *addition, subtraction, multiplication, division, and modulo*. For floating-point numbers, the same operators excluding the modulo operator are available.

2.3 Control Structures

2.3.1 Conditionals

Conditionals in Eridu works on it's primitive types as one would except from a typical programming language. The result of a conditional statement is an integer, which gets cast to either 0 or 1 depending on the truthiness of the statement. Not all of these operations are available to all primitives, *char* and *str* values can only have their equality checked using == and !=. For pointers, which will be described further in section 2.6, a direct comparison will lead to a comparison of the address pointed to, treated as an integer.

```
a < b  // less than
a > b  // greater than
a == b // equality
a != b // inequality
a <= b // less than or equals
a >= b // greater than or equals

int b = 5 > 4; // b = 1
```

Fig. 1: Example of Conditional Operators in Eridu

2.3.2 Loops

The Eridu language only provides while loops as a compile-time feature. We reasoned that only this type of loop would be necessary, as it can be used to implement other types of loops such as for-loops. Below we give an example of the usage of the while loop.

```
int b = 10;
print(b); // 10
b = reduce_to_one(b);
print(b); // 1

define int reduce_to_one(int a ){
    while (a != 1) {
        a = a - 1;
    };
    return a;
};
```

Eridu also provides a break and a continue statement. We provided these as we found them to be important for adding exceptions and additional program control, an example of this can be seen on figure 2.

```
int i = 10;
while (i > 0) {
    print(i); // prints 10, 9, 8, 7, 6
    if (i == 5) {
        break; // breaking from while loop
    };
    i = i - 1;
};

print(i); // prints 5

i = 10;
while (i != 0) {
    i = i - 1;
    if (i == 6) {
        continue; // continuing in while loop
    };
    print(i); //prints 9, 8, 7, 5, 4, 3, 2, 1, 0
};
```

Fig. 2: Example of Break and Continue

2.3.3 Blocks

Blocks can be declared to create new scopes. These scopes count as an inner-scope of the scope at the point they are declared. Eridu uses static nested scoping using static-link chaining, such that an attempt to access a variable in an outer scope will only be successful if the scope is accessible from the current scope.

```
int a = 5; // A will belong to the global scope
{
    print(a); // output: 5
    a = 10;   // global scope a = 10
    {
        int b = 5; // b & a accessible in this scope
        // but b is not accessible in the parent scope

    };
};
```

If a re-declaration of an outer scope variable is performed in an inner scope, the re-declaration will capture immediately upon entering the scope, regardless of where the declaration was made in the inner scope. As a result of our function implementation discussed in section 2.9.1, there are some important details to consider when using outer scope variables. Specifically, calling a function, which uses an outer scope variable, in another scope than the one it was declared in will result in undefined behaviour.

2.4 Functions

As with most programming languages, functions exist in Eridu as a way to organize- and facilitate reuse of code. Each function has its own scope, and as previously mentioned, can walk up its static nested scope and access elements therefrom. Using syntax similar to typical C-like languages, we show a simple function definition on figure 3.

Similarly, we allow for recursion, but unlike languages such as C, we further allow developers to define inner-functions. We wanted this feature as we noticed especially while developing this compiler, that the entry point to a recursive function typically has to perform setup of some kind, often resulting in two function definitions, one for the entry point and one for the recursive logic itself, unnecessarily populating the name space of a given scope. Here it is important to note that the inner-function will *not* be redefined each call to its outer function, as is the case with languages such as Python[1]. Further it should be noted, that the use of inner functions also serves as a method of *encapsulation*.

```
// Set debug flag
int debug = 1;

define int agecheck(int age) {

    if(age >= 18) {

        if(debug == 1) {
            print("Is an adult");
        };

        return 1;
    };

    if(debug == 1) {
        print("Is not an adult");
    };

    return 0;
};
agecheck(18); // Is an adult
```

Fig. 3: Simple function definition and static-scope accessing

2.5 Intrinsics

Before proceeding to composite types, we will discuss the intrinsic functions native to Eridu. In compiler theory, intrinsics refer to built-in functions handled at the compiler level. In languages such as C, intrinsics come in many categories, some map directly to x86 instructions, some to SIMD[2], and some function primarily as an optimization for functions which could be used by linking to a library containing an implementation of said function. An example of a C intrinsic is the *strncpy* function, for which a library implementation would result in the creation of a new stack frame, whereas the compile-level implementation can generate the corresponding assembly instructions in the place it is needed, without the overhead of the former.

```
struct ExampleStruct = {
    int a,
    char b,
    float c,
};
int size = sizeof(
    struct ExampleStruct
);

print(size); // 3
```

Fig. 4: Use of the *sizeof* intrinsic function

[1]To be clear, in Python a new function object is created each call, but the underlying code is re-used.
[2]Single instruction, multiple data

We decided to keep the number of compile-time functions to a minimum, as it adds extra complexity to a project where the value of optimizations should be weighed against our limited time frame. As seen on figure 4, one of the functions we implemented was *sizeof*. In section 5.5.2 it will be mentioned that all primitives in Eridu are *currently* implemented as 64bit quad-words. Because of this, *sizeof* is currently only interesting with regards to structures, as it counts the number of member fields.

While this seems to conflict with our previous statement of weighing complexity against project scope, this came about as this was a learning experience for us, and found that we in fact did not want to spend extra time implementing multiple sized types later in the projects time frame. More relevant intrinsics for the current stage of development are the *cast*, *allocate*, *dereference*, *addressof*, and *print* intrinsics, all of which we briefly describe as follows:

```
// Allocate 10 ints
int * array = allocate(int, 10);

// Allocate 10 pointers to ints
int ** matrix = allocate(int *, 10);

// Allocate 1 int
int * ptr = allocate(int , 1);

// Set the value of the int to 1
dereference(ptr) = 1;

print(dereference(ptr)); // 1

// Get the address of a variable
int a = 5;
int * ptr = addressof(a);
```

Fig. 5: Use of the *allocate* and *dereference* functions

Allocate, Addressof & Dereference; As shown on figure 5, these built-in functions provide the ability to allocate memory, and write or read to and from these addresses again. We decided on giving *allocate* literal types, instead of a number of bytes as seen in C, as we have no concept of *void pointers* in our language, and thus developers should always know which type of data they want to store.

Cast & Castable; These two methods allow developers to work around our strict type system, as types such as *int* and *float* are not implicitly typecast when using operations such as general arithmetic, but rather have to be explicitly cast. Here it is important to mention, not only in this thesis but also to possible developers that casting floating points to integers works by *rounding down* to the nearest integer

Print; As would be expected, print outputs its given data to *stdout*, not much more is to be said about this method, except that in contrast to most of the language, *print* is actually type invariant, being able to print strings, characters, integers and floats without specifying the type of said values. We made this design choice as *print* has no implications on the logical operations of a program, and thus, implicit typing can not cause logical errors such as unexpected results of arithmetic procedures reasoned by implicit type casting.

2.6 Pointers

As can be seen on figure 5, Eridu provides pointers as an explicit type to the user. The syntax of their declaration is as with other C-like languages. We chose to allows explicit pointers in Eridu, as we feel they give a lot of control to programmers experienced in their use, such as pass-by-pointer parameters for functions or in-line modification of composite types. Also, as will be described in the subsequent section on composite types, they are paramount for Eridu's composites.

2.7 Composite Types

Composite types exists exclusively on the heap in Eridu. Therefore, for any access to these, a pointer must be made to a heap allocated block of memory.

All composite types stored within another composite type must be "unpacked" into a variable of the same type to be used. The reason for this is a result of the time available for the project. When the first iteration of composites were complete, we did not have enough time for a proper generic implementation of nested composite-type referencing. While this feature would have been nice to have, we feel that since the same programmatic capabilities are available without, we would create more value by focusing on polishing the features we already had.

2.7.1 Arrays

Arrays in Eridu stores homogeneous data like in most C-like languages. As with C, an array index can be the result of a run time expression. Arrays of all primitive types, or other composite types can be made using the allocate intrinsic function and pointers.

```
int * arr;
int size = 5;
arr = allocate(int, size);

int i = 0;
while (i < size) {
    arr[i] = i;
    print(arr[i]); // prints 0, 1, 2, 3, 4
    i = i + 1;
};
```

Being able to store data as an array comes with obvious benefits to the programmer. Tasks such as sorting large amounts of data becomes much more viable in Eridu provided these arrays.

2.7.2 Structs

Data structures in Eridu works similarly to C, using the following syntax for declarations.

```
struct MyStruct = {
    int a,
    float b
};


struct MyStruct * data = allocate(struct MyStruct, 1);

data.a = 5;
data.b = cast(float, data.a);

print(data.a); //prints 5
print(data.b); //prints 5.00...
```

The benefit of having data structures in Eridu are immense. This allows the storage of heterogeneous data., other composites can also be stored in data structures, allowing for more complex ways to structure data.

2.8 Command-line-argument input

Command-line arguments can be accessed in Eridu by first declaring the following variable in the global scope:

```
str * arguments;
```

This tells the compiler that the user wishes to access the list of command-line arguments, which a user can then index into. This str pointer allows developers to access command-line arguments one-by-one. This works in a similar way to the C construct "argv", with the first index containing the name of the run program.

The usefulness of command-line arguments can be illustrated in the example seen on the next page. In this example, we read a size to an array, and a comma separated integer array in the form of strings from the command-line input. We then convert the the first command-line string into an integer, indicating the size of the input array. From the second argument, we get a comma separated array in the form of a string, which we then load number by number into an integer array. Finally, the given input array is printed from the int array storing it.

```
str * arguments;
str input = arguments[2];
int input_size = number_string_to_int(arguments[1]);
int * array = fill_array_from_input(input, input_size);

int i = 0;
while (i < input_size) {
    print(array[i]);
    i = i + 1;
};

// Takes a comma-seperated input array
// in the form of a string, and converts
// it to a valid Eridu int array
define int * fill_array_from_input
(str input, int input_size) {...};

define str get_next_num(str comma_number, int index) {...};


// Takes a string and converts it to an integer
// if possible, otherwise, behaviour is undefined
define int number_string_to_int(str number)  {...};

define int pow(int num, int power) {...};

// Looks for an end-of-line char in the string,
// incrimenting an integer for each index
// reached before the end-of-line char
define int string_length(str string) {...};


// Takes in a char and converts it to
// a single-digit integer if possible,
// otherwise returns -1
define int digit_string_to_int(char digit) {...};
```

2.9 Functions as Variables

In sections 2.4 we gave an example of a simple function definition in Eridu, however, we wanted to expand on the concept of functions, allowing them to be used in a more dynamic way by developers. To achieve this, we introduce the *callable* type. Callables are strictly typed references to functions, that can be used as regular variables. The use of callables in conjunction with functions will be described in section 2.9.1, while in this section we take a look at the synergies between callables, structures and arrays. Easing into this concept, we start by showing an example of callable definitions and their type signatures on figure 6

```
// Misc.
define int _add(int x, int y) {...}
define int _sub(int x, int y) {...}

// Declare a Callable
<int> callable <int, int> operation;

operation = _add;
print(operation(2,1)); // 3

operation = _sub;
print(operation(2,1)); // 1
```

Fig. 6: Callable definition and type signature

As will become clear in section 2.10, having references to functions which can be switched around during run time, allows developers to use structures and arrays in ways much greater than the sum of their parts. But before delving too deep into the more complex abstractions that this enables, let us first take a look at how both arrays and structures can hold callables, have callables freely reassigned, and how they can be used in a program later on. This is shown on figure 7, and a less arbitrary example of this can be found in the appendix on figure 45.

```
struct Operator = { <int> callable <int, int> op, };

<int> callable <int, int> * functions
    = allocate(<int> callable <int, int>, 2);

functions[0] = _add;
functions[1] = _sub;

struct Operator * my_operator = allocate(struct Operator, 1);
my_operator.op = functions[0];

// Can later be used like so:
operation = functions[0];
print(operation(4, 1)); // 5

operation = functions[1];
print(operation(4, 1)); // 3

operation = my_operator.op;
print(operation(5, 2)); // 7
```

Fig. 7: Callables and composite types

2.9.1 Higher-Order Functions

This notion of callables means that Eridu can treat functions as first-class citizens. This differs from first order functions which in contrast, can not have functions passed as arguments, nor can they be returned from a function. Typically when languages feature higher-order and nested functions, they utilize *closures* to encapsulate the environment a function was defined in, binding names from a lexical scope and storing them in a record bundled together with the callable itself, which allows a function to access said environment when called or used elsewhere [1, Ch. 15.2, p. 329].

Eridu does not implement the concept of closures, and thus, we look at nested functions as isolated entities when they are called in scopes that can not be statically linked to the scope where they were defined. This design decision was primarily reasoned by the scope of the project, as we implemented a wide array of features and found that closures was not a priority with regards to the fundamental usability of the language.

More examples specifically regarding the ability to pass callables as arguments will be described in section 2.11, but to showcase how higher-order functions can work together with other features of the Eridu language, we show on figure 8 how to assess user input and use it to perform operations depending on control flow evaluated at run time. In the next section we will take a look at how these concepts can be used to approximate behavior seen in other programming paradigms, without actually implementing most of the properties that these paradigms are built around.

```
define <int> callable <int,int> get_calc_op(str name) {
    define int modulo(int x, int y) { return x % y; };
    define int addition(int x, int y) { return x + y; };
    define int subtraction(int x, int y) { return x - y; };

    if(strcmp(name, "modulo") == 1) {return modulo;};
    if(strcmp(name, "addition") == 1) {return addition;};
    if(strcmp(name, "subtraction") == 1) {return subtraction;};

    // Default
    return addition;
};

// Initialize Arguments
str * arguments; // ["program.o", "modulo", "10", "3"]

define int main() {

    // Get operation based on input
    str operation = arguments[1];

    // Get arguments
    int x = number_string_to_int(arguments[2]);
    int y = number_string_to_int(arguments[3]);

    // Get operation function
    <int> callable <int, int> operator = get_calc_op(operation);

    // Print result
    print(operator(x,y)); // 1
    return 0;
};
main();
```

Fig. 8: Dynamically return inner functions based on user input

2.10 Approximating Object-Oriented Programming

The ability to organize a program around concepts and objects, rather than functions and logic, allows programmers to work with more abstract entities, making it possible to create separation between the implementation of a program and the responsibility of said program. These abstractions are widely used, especially in object-oriented programming, and are often referred to as *design patterns*. Many of these patterns rely on the four fundamental object-oriented programming concepts: inheritance, encapsulation, data abstraction, and polymorphism, and thus, are not always easily implementable in languages not adhering to the object-oriented paradigm.

The use of design patterns can play a large part in maintaining code and preventing what is commonly known as code-rot, as code can depend on abstractions rather than implementations, reducing the likely hood of a change causing dependent code to break. As reasoned by these benefits of the object-oriented paradigm, it was important for us to ensure that our language did not waive the option to use some of these concepts, without straying away from the philosophy of our language. We here showcase how the iterator, singleton, and the facade patterns can be implemented, relying primarily on the synergy between structures and functions, without the need for distinct object-oriented programming language features.

2.10.1 Iterators

The Iterator pattern is a behavioral pattern, often used when the traversal of a collection of elements should be wrapped in an abstract method, allowing developers to separate the implementation of the method, from the outcome of the method [2, Ch. 5, p. 249, p. 289].

One of the substantial benefits of the iterator pattern is the ability to dynamically change the definition of the order in which a collection should be traversed, while maintaining *loose coupling*. To implement an iterator in Eridu, we first define a structure which stores the information to iterate over, an example of which can be seen on figure 9.

```
struct Iterator = {
    str * values,
    int    size,
    int current,
    <str> callable <struct Iterator *> next,
};
```

Fig. 9: Definition of an Iterator Structure

Within this iterator, besides from storing user-defined data, we can also store references to functions that define the behavior and definition of some traversal. This could be expanded to create bidirectional iterators, iterating a binary search tree, or used to filter elements based on some criteria. To showcase the use of the iterator pattern, we refer to figure 10, which shows the ability to change the implementation of the *next* callable.

```
// String Iterator
struct Iterator = {...};

// Typical linear next function
define str _next(struct Iterator * iter) {...};

// Next is the first element larger than the current
define str _next_larger(struct Iterator * iter) {...};

define struct Iterator * create_iterator(
             char ** values, int size) {...};

define int main() {

    // Create dummy data
    str * names = allocate(str, 5);
    names[0] = "Michael";
    names[1] = "Peter";
    names[2] = "Hamurabe";
    names[3] = "Jackson";
    names[4] = "Wolfeschlegelsteinhausenbergerdorff";

    // Default iterator is a linear iterator
    struct Iterator * iter = create_iterator(names, 5);

    callable next = iter.next;
    str a = next(iter);
    print(a); // Michael
    a = next(iter);
    print(a); // Peter

    ...

    // Could be implemented as a "reset" function.
    iter.current = 0;

    // Get the next name larger than the current
    iter.next = _next_larger;

    // Unpacking callable
    next = iter.next;
    a = next(iter);
    print(a); // Hamurabe
    a = next(iter);
    print(a); // Wolfeschlegelsteinhausen...

    return 0;
};
main();
```


Fig. 10: Practical use of the Iterator Pattern

As mentioned, the Iterator pattern is behavioral pattern, differing from the two other primary groups, creational and structural patterns. Creational patterns refer to patterns whose primary objective is to make the instantiation of objects abstract, aiding in the overall decoupling of a project and helping developers create independent systems[2, Ch. 3, p. 94].

2.10.2 Singletons

An example of these creational patterns is the singleton pattern, which has to purpose to ensure that a class can only ever have a single instance of it made[2, Ch. 3, p. 144]. Benefits from this become clear when you consider objects such as window managers or file systems, as typically only a single instance of these objects is desired.

But how does this differ, from creating an instance of an object in the global scope, and having developers refer to this object? well, as a way of reducing memory, and handling cases where developers are trying to access an object, that *might* get created depending on the control flow of a system. On figures 11 and 12 we show the implementation of the singleton pattern in the Eridu language.

```
struct Logger = {...};
define struct Logger * construct_logger(){...};

struct LoggerSingleton = {
    int has_singleton,
    struct Logger * logger,
    <struct Logger *> callable<...> constructor,
};

define struct LoggerSingleton * make_singleton() {
    struct LoggerSingleton * this
        = allocate(struct LoggerSingleton , 1);

    this.has_singleton = 0;
    this.constructor = construct_logger;
    return this;
};
```

Fig. 11: Setting up a Singleton for a Logger Object

Having now created the required structures, we can now imitate the effect of a singleton by creating a *getter* function, which can be made accessible to developers in the global scope, while avoiding the memory and control flow issues previously mentioned.

```
define struct Logger * get_singleton(
        struct LoggerSingleton * singleton) {

    if(singleton.has_singleton == 1) {
        print("Singleton already has a logger");
        return singleton.logger;
    };

    // Unpack Callable
    <struct Logger *> callable c = singleton.constructor;

    // No singleton exists, make one.
    singleton.logger = c(...);
    singleton.has_singleton = 1;

    return singleton.logger;
};

// Globally accessible singleton
struct LoggerSingleton * singleton = make_singleton();

// Ensure that the singleton was created exactly once
// and returns the same object every time.
struct Logger * logger1 = get_singleton(singleton);
struct Logger * logger2 = get_singleton(singleton);

// Ensure their memory addresses are the same
if(logger1 == logger2) {
    print("Singleton works!");
};
```

Fig. 12: Implementation of a Singleton Object for a Logger

While this might seem verbose, it is important to note that much of this code would also exist in object-oriented implementations of a singleton, usually encased inside the class definition of the logger itself.

2.10.3 Facade

Lastly, we showcase the *Facade* pattern. Facades provide a unified interface from a set of interfaces. This unification can help reduce the complexity of systems, lessening its dependencies by way of loose coupling, and decrease communication between the users of a system, and the implementation of said system[2, Ch. 4, p. 208]. We begin by defining the set of interfaces we want to abstract on figure 13.

```
struct Lights = {
    int is_on,
    <int> callable<struct Lights *> off,
    <int> callable<struct Lights *> on,
};

struct Projector = {
    <int> callable <struct Projector *, int> play_movie,
};

struct Speaker = {
    int volume,
    <int> callable <struct Speaker *, int> play_track,
    <int> callable <struct Speaker *, int> set_volume,
};
```

Fig. 13: Various structures defining a Theater system

Having now loosely defined the subsystems which make up a theater, we can implement a facade, seen on figure 14, which wraps larger scale logic in a simpler interface.

```
struct TheaterFacade = {
    struct Lights * lights,
    struct Projector * projector,
    struct Speaker * speaker,

    <int> callable<struct TheaterFacade *> start_movie,
    // Futher expanded: set_price, sales_report ...
};

define int _begin_movie(struct TheaterFacade * this) {
    // Turn of Lights
    struct Lights * lights = this.lights;
    callable off = lights.off;
    off(lights);

    // Play Movie
    struct Projector * projector = this.projector;
    callable play = projector.play_movie;
    play(projector);

    // Play Soundtrack
    struct Speaker * speaker = this.speaker;
    callable soundtrack = speaker.play_track;
    soundtrack(speaker, 6);

    // Set Volume
    callable set_volume = speaker.set_volume;
    set_volume(speaker, 60);
    return 0;
};

define struct TheaterFacade * make_theater(){
    struct TheaterFacade * this
        = allocate(struct TheaterFacade , 1);

    this.lights = make_lights(...);
    this.projector = make_projector(...);
    this.speaker = make_speakers(...);
    this.start_movie = _begin_movie;
    return this;
};

// Facade now allows simple access
// To a more complex system.
struct TheaterFacade * theater = make_theater();
callable start_movie = theater.start_movie;
start_movie(theater);
```

Fig. 14: Theater Facade Pattern

Whilst it seems like a developer could just as well use the *_begin_movie* function directly, consider the previously mentioned design patterns. As with iterators, this facade can easily switch the implementations of say, its speaker, if a new one was bought whose interface differed from the previous one, without the users of the facade even knowing. Furthermore we can use the concept of the singleton, to ensure that only one theater exists, avoiding the use of older instances which have not implemented newer functionality.

As our concluding remarks regarding the approximation of the object-oriented paradigm, we see that the use of functions as variables, combined with structures, is an incredibly powerful tool allowing the use of at least some design patterns commonly known from object-oriented programming.

2.11 Unlocking the Functional Programming Paradigm

While Eridu does not support common functional programming concepts such as currying and partial evaluation, it is possible to imitate some features of impure functional languages. This ability stems from the implementation of higher-order functions as described in section 2.9.1, and since functions in Eridu are treated as first-class citizens, we can create functions which takes functions as arguments, and return functions as well. While our language does have the notion of state, and does not strictly enforce ideas such as immutability and function purity, it is still possible to implement some typical functions such as map, filter, and reduce, which are often seen in functional language

Further, it is also possible to define inner functions and return these functions dynamically as shown on figure 8, however as mentioned in section 2.9.1, these functions do not encapsulate their surrounding environment by way of closures. Based on the former description of the capabilities of our language, it would not be correct to call Eridu a functional language, but rather an imperative language in which developers can choose to utilize some fundamental concepts.

In this section, we will describe and showcase the implementation of the map, filter, and reduce functions. In figure 15 we see an example of the *map* function, which in this case, adheres to the idea of self-containment, implying that no outside state affects the execution of the function, nor are changed by the function. Further we avoid modifying the given input by allocating a new array, imitating immutability.

```
define int * map(int * arr, int size,
    <int> callable <int> func) {

    int i = 0;
    int * new_arr = allocate(int, size);
    while(i < size) {
        new_arr[i] = func(arr[i]);
        i = i + 1;
    };

    return new_arr;
};
```

Fig. 15: Implementation of Map in Eridu

While map might seem like a simple function, it gives developers the ability to write powerful code while being less verbose, and reduce the overall burden of performing generic tasks. As a side effect, map also allows us to emulate for-each loops, since these are not implemented in the language for reasons mentioned in section 2.3.2.

Similarly to map, another common function seen in this paradigm is the *filter* function, which takes in an array of elements, along with a function used to evaluate whether or not an element of said array should be returned or not, as its concept is quite simple, we refer to both it, and the implementation of the *reduce* function, which simply folds a given array into a single element using a function such as *sum*, on figures 16 and 17.

```
// if size < new_size, all newly allocated
// slots will be initialized to 0
define int * realloc(int * arr, int size, int new_size) {

    int * new_arr = allocate(int, new_size);
    int i = 0;
    while(i < size) {
        new_arr[i] = arr[i];
        i = i + 1;
    };

    return new_arr;
};

define int * filter(int * arr, int size,
    <int> callable <int> cond) {

    int i = 0;
    int j = 0;

    int * new_arr = allocate(int, size);

    while(i < size) {
        if(cond(arr[i]) == 1) {
            new_arr[j] = arr[i];
            j = j + 1;
        };
        i = i + 1;
    };

    return realloc(new_arr, size, j);
};
```

Fig. 16: Implementation of Filter in Eridu

```
define int reduce(int * arr, int size,
    <int> callable <int, int> func, int init) {

    int i = 0;
    while(i < size) {
        init = func(init, arr[i]);
        i = i + 1;
    };

    return init;
};
```

Fig. 17: Implementation of Reduce in Eridu

3 Parsing

3.1 Tooling

Early on, we decided on not implementing our own lexer and parser for the development of our compiler. While we did have an interest in experiencing the development of a compiler from end-to-end, it felt more appropriate to focus our time and energy on developing interesting features. Our current tool chain and its implications on our compiler will later in this section be briefly discussed, but in the future, we definitely want to implement these phases ourselves. In part because we could tailor them to fit our specific needs, removing the overhead which these tools bring, as we have no need for many of the complex features which they support.

3.1.1 (F)LEX

To tokenize input programs to the Eridu compiler, we chose to use the lexical analysis tool *Flex*. Flex lets developers write POSIX regular expressions which it then in turn compiles into C code which can perform tokenization based on these rules. Being able to purely focus on the regular expressions themselves, and not the logic required to iterative through a given source file, greatly increases the ease and flexibility of the development process, as changing the functionality of this stage is quite easy.

3.1.2 Bison/YACC

As with FLEX, we also decided to use the parser generator *Bison*. Bison lets developers write abstract grammars and to add regular c-programming behaviours to each rule of the grammar. We took advantage of this to build our abstract syntax tree while parsing the input code. Bison by default generates an LALR(1) parsing-table for the grammar which was defined, and lets us as developers avoid making a new parsing table each time we modified the grammar, and instead put more energy into improving later stages.

3.2 Implications

Using this set of tools, while making it easier to modify our codebase, also slightly restricted some aspects of the compiler, which might have decreased the overall compile time. However, these tools were what allowed us to focus most of our energy on the stages we deemed the most important.

3.3 Grammar

One of the earliest steps of the design process of this project, was to design and formalize a grammar describing the Eridu language. We did this in an iterative fashion, adding productions as we implemented new features, as we felt that this would cause the least interference and confusion in the development process. Our first step in the design phase was to look at the grammar of languages such as C, to get an idea of how the abstractions and recursive definitions of constructs were set up. Admittedly, we ended up with a grammar which could be optimized or polished a bit, as a result of our iterative implementation of productions, but we did end up with a grammar which fulfilled our needs, and adhered to our desired precedence- and structural requirements. A few excerpts of our grammar can be seen on figure 18.

```
expression : literal
    | type
    | functioncall
    | expression STAR expression
    | expression NEQUAL expression
    | expression LEQ expression
    | expression GEQ expression
    | expression EQUAL expression
    | expression '>' expression
    | expression '<' expression
    | expression '+' expression
    | expression '%' expression
    | expression '/' expression
    | expression '-' expression
    | %empty

simpletype : INT
     | STR
     | CHAR
     | FLOAT

typedcallable : '<'type'>'CALLABLE'<'typelist'>'
              | '<'type'>'CALLABLE
              | CALLABLE

pointer : type STAR

type : simpletype
     | typedcallable
     | pointer
     | STRUCT ID
```

Fig. 18: Caption

4 Type System

Type systems refer to some logical system which purpose is to assign *'type'* properties to various constructs. In less abstract terms, these systems help both formalize and categorize the behavior of constructs such as those used for arithmetic purposes, grouping data like composite types, or defining the return value of functions, and are put into place primarily as a way of diminish the prevalence of errors in program. When these systems are in place, they can be used to help evaluate whether or not a given expression uses terms - which have now been given type categories - correctly. By looking at the structure and flow of a program, the types of constructs on either side of an expression can then be compared, validating their compatibility with not only each other, but also how they are used in a more abstract sense.

Previously in section 1.1, we touched on which type of programming environment we wanted our project to be conducive to, and at the core of this philosophy sits our type system. As there are quite a lot of nuances regarding type systems and the design choices which we had to make, we attempt to describe each of these in the following paragraphs:

Static type checking A type checking phase is classified as being *static*, if it performs its validity and safety checks at compile time, usually by way of analysing the source code of a given program. Before attempting to properly explain the implications of static type checking, it would be best to briefly discuss another commonly used method first; *dynamic* type checking.

Dynamic type checking refers to systems which instead of assigning types to constructs during the compilation stage, this approach usually assigns a reference to some type object with specific information about said type and its properties, to each run time object. In contrast to static typing, this approach can not validate program correctness before run time, making it entirely possible that a program can behave in an expected manner, but suddenly terminate as a result of faulty code which prior to the current state of control flow, had not been executed. Using dynamic typing, it may not be possible to verify the correctness of a program as well as one could with static typing, but some language features such as down casting can not be statically checked, and therefore many modern languages use both of these methods in conjunction with each other.

While we were in the initial design and research phase of this project, we made the decision to make our language purely statically typed. One of the reasons behind this decision was our inspiration from C, as mentioned in 1.1 we in many ways wanted to develop a language very similar to C, and attempt to personalize it. More importantly however, we also felt that a statically typed language was the most feasible considering our prior knowledge of the topic, and concluded that we it would be more interesting and a better fit regarding our project timeline.

Manifest Typing One of the more obvious choices a programming language makes from the point of view of the developer, is to which extent a language features implicit typing. In many languages such as JavaScript and Python, variables need not always be explicitly declared with a the type of value it is referencing to. This system relies on the concept of inference, where the compiler or run time process attempts to deduce which kind of value is being used by context, and systems like these can carry with them benefits such as reduced development time and less verbose code.

However downsides such as as slower compile times as type inference requires additional computation, and making it harder to intuitively understand how the compiler interprets and uses code, does not fit with the ideals of the language we wanted to develop. In the same vein, we felt strongly that the unsafe development environment which implicitly typed languages can create by the freedom to do more while knowing less, was the opposite direction of where we wanted to go, and as such, we decided on what is called *manifest* typing. At its core manifest typing simply requires explicit type declarations for constructs such as variables, and thus also forces developers to have a better understanding of what they are writing.

Having decided on implementing a purely statically-typed language featuring manifest typing, we will now briefly introduce the concepts which we used internally in the compiler to orchestrate our type system.

4.1 Type Symmetry

In addition to being statically and explicitly typed, Eridu also uses the notion of symmetrical typing. In practical terms this means that the type of the left-hand side of an expression, should be identical to the type which the right-hand side evaluates to. This becomes clear when we as mentioned in section 2.5 look at casting in Eridu.

Many explicitly typed languages still feature some variants of inferred typing, in C this is seen in *coercion's*, which occur when say, developers attempt to perform simple arithmetic using both an integer and a floating point value. Type coercion's cast values to other types implicitly, removing the need to explicitly cast or handle type conflicts in many smaller situations where it would seem cumbersome or redundant to do so. However, for this project we decided to stick to a very strict definition of explicit typing.

The primary implication of this decision, was rendering the actual implementation of our type checker relatively simple, as we for most constructs in our abstract syntax tree could fetch the type of each child node and compare their equality to assert whether or not it was valid. While our type checker in practice performs many more steps to ensure the semantic validity of a program, this is the most essential concept on which it is built.

4.2 Callable Inference

Strongly contradicting our previous statements, we did want to implement an all be it weird, but also interesting concept which would allowed developers to break the mold. As we mentioned in section 1.1, we found it important to let developers assume full responsibility of a programs correctness, and as such, we introduce the concept of *untyped callables*. In section 2.9 we showcased the concept of *callables*, and how they let developers store references to functions as were they variables. In addition to this, it is also possible to declare a callable without explicitly declaring parameter and return types.

```
define int add(int a, int b) {
    return a + b;
};

callable a = add;
int c = a(1, 2);
print(c); // 3
```

Fig. 19: Example of Untyped Callables

While this decision seems to go against most all of our previous ideals, we took advantage of the fact that this untyped concept is not at the core of our language, allowing us to use it as a platform to learn and experiment without implicating features whose safety and intuitiveness is essential.

Breaking from our strict sense of explicit typing, untyped callables use type inference to slip through the type checking phase, essentially imitating the type of the left-hand side of an expression wherein they are used. This is not a safe procedure, and thus, no guarantees can be made that errors during run time will not occur from their use.

Ad Hoc Polymorphism One of the more interesting ramifications of this concept, is that it can enable a very informal variant of ad hoc polymorphism. Since neither the parameters or return values are typed, one can call one of these callables with any number or type of arguments and naively assume that the return value of the function call will be the same as your left-hand side expression. Because of this, a daring developer with a keen understanding of how our compiler works internally, can develop solutions of arguable sanity. On figure 20 we show an example of this type check bypassing being used to interpret characters as integers corresponding to their ASCII values.

```
define int add(int a, int b) {
    return a + b;
};

callable a = add;
int c = a('c', 4);
print(c); // 103
```

Fig. 20: Untyped Callables used to bypass Type Checking

Another implication of its type-less property is the passing of arguments. If we take another look a figure 20, and imagine we passed in yet another argument, the integer *6*, we would actually get a return value of *10*. Untyped callables utilize the exact number of arguments the function definition it aliases expects. This also means that if too few parameters were given, the program would read garbage during run time. The exact reasons as for why untyped callables use the last n arguments given instead of the first n arguments are a result of specific implementation details of our parameter passing, and as this is in no way intended as a core feature of our language, we will not go into further detail about untyped callables. We will however mention that this concept has served as an important part of our development, as it made for a great tool for performing edge-case testing and gave us quick access to the phases of compilation proceeding the type checking stage.

4.3 Array Equivalence

Circling back to concepts more fundamental to our language, we take a look at how the equality of types are actually compared. This is generally not an issue when looking at primitives, as objects of the same primitive type only vary in their contents, and not their properties. The same can not be said for composite types however, as both arrays and structures can be defined to store any type of value. If we first look at arrays specifically, we decided on using the notion of *strict* equality, meaning identical, in contrast to *abstract* or *loose* equality where the it is not the instance, but the values that are tested.[3] In practical terms, this means that arrays in Eridu are compared using their literal memory address, rendering the contents of arrays irrelevant with regards to their equality.

[3]The meaning of these terms in this context, is identical to that of the JavaScript MDN documentation. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness

```
define int compare_arrays(int * a, int * b, int size) {
    int idx = 0;
    while (idx < size) {
        if (a[idx] != b[idx]) {
            return 0; // Arrays are not equal
        };
        idx = idx + 1;
    };
    return 1; // Arrays are equal
};
```

Fig. 21: Example Implementation of an Array Comparison Function

While this is a simple solution, we made the decision as implementing a function to compare arrays is trivial, allowing developers greater configurability with regards to their own definition of equality. This solution fits nicely with our ideal of writing a simple language on which developers can build more complex logic, and a simple implementation of an array comparison function can be seen on figure 21.

4.4 Structure Equivalence

Nominal equivalence describes a very strict form of equality, where the name of the structures - the type, not the instance - has to be the same. In practical terms this means that no structures can be the same, unless they are based on the same *struct* definition. This is in contrast to structural equivalence, where in this case structures, are equivalent if they have the same general structure. This equivalence system has a variety of different levels of strictness which one can adopt, but commonly if two structures has the same number, type, and order of fields, structures are seen as being equivalent.

Eridu utilizes nominal typing, as we felt that structural typing made little sense, as our type system is meant to be strict especially regarding the type of arguments and return values from functions. If a function takes in a specific structure type as an argument, having different types of structures being equivalent has very few practical uses. Further, if we did decide on structural typing, since our type system is symmetric with regards to the specific type of the left- and right-hand sides invariant of equality, it again limits the practical use of structural equivalence.

```
struct MyStruct    = { int a, int b };
struct TheirStruct = { int a, int b };

struct MyStruct * my_struct = allocate(struct MyStruct, 1);
struct MyStruct * my_struct_2 = allocate(struct MyStruct, 1);
struct TheirStruct * their_struct = allocate(struct TheirStruct, 1);

if (my_struct == my_struct_2)  {
    print("Equivalent"); // Equivalent
};
if (my_struct == their_struct) {
    print("Not Equivalent"); // Did not trigger
};
```

Fig. 22: Example of Nominal Typing for Structures in Eridu

On figure 22 we see the nominal equivalence system in action, notice that both structure definitions are identical, but still are not equivalent. However one interesting question does appear. Some implementations of structural equivalence differs in strictness from each other, some variants actually care about the name of fields, which could be interesting as it would ensure that they can be accessed using the same identifiers. This would mean that developers could use equality between structures to verify that they can be accessed in a more generic manner, but still, inconsistencies between structural typing and our language philosophy appear. Maybe the most interesting for us to look at is from a developers point of view, say two structures have identical structures, but the developers *intentions* for said structures are different, it would not make sense to call them equivalent, as in the developers mind, they are not.

However, should the need for structural equivalence arise, we did actually implement a variant of it which we could plug into our type checker, should we ever want to tinker with its implications in our language.

5 Back-End of the Compiler

The Eridu Back-End is made up of two separate stages consisting of the Codegeneration stage and the Emit stage, each of which can be quickly summarized thusly:

- The Codegeneration stage traverses the AST and generates intermediate representations corresponding to each node.

- The Emit stage translates these intermediate representation instructions generated by the previous stage to literal x86-64 assembly in AT&T syntax and outputs it into a file called "output.s".

5.1 The Context-Object Status

To keep track of our instructions and their order, we use a Context-Object called Status. Status contains a Linked-List of pointers to INSTRUCTION structures, which we use internally to represent our intermediate representation.

The instructions are created in the order of the program flow from the AST, except in special cases where left and right nodes must be recursively evaluated before the parent node, such as an arithmetic node, where we want to evaluate the sub expressions first, before we combine the results using the operator of the arithmetic node. Our status structure also helps keep track of symbols by storing a pointer to the current symbol table we are working on. We use this to keep track of scoping during code generation, as the symbol table is updated upon entering and leaving scopes.

In addition to this, our context-object also stores labels with data for string and float literals created during code generation. This lets the compiler emit all these labels, along with the corresponding data, to the data segment of the assembly file.

5.2 Floating-Point Numbers & String literals

Floating-point numbers are essential for a language to provide mathematical and scientific calculations. In Eridu, this is provided using the x87 Floating-Point operations and registers (SSE registers). By doing this, we take advantage of modern CPU's ability to process these kinds of numbers, and also simplify the rest of the compiler, by not having to specify our own byte-level implementation. We get float literal values by adding an appropriate label to the data section of the assembly file and use the .float keyword to store it as a valid x87 floating-point number.

Strings works in much the same way, with literal strings being added to the data segment using the .string label. This effectively allocates a set of concurrent character addresses on the heap, before the beginning of the modifiable heap section. The implications of this will become clearer when we discuss heap memory in section 5.7.

5.3 Operators & procedures

Most operators and procedures follow a relatively simple pattern. Any operands are first pushed to the stack before the start of the procedure or operation. At the end of a procedure or operation, the result is pushed to the stack. Thus, the result of previous operations or procedures can be the operands of subsequent operations and procedures.

One of the main invariants of the compiler, is that the result of all operations or procedures are stored directly on the stack. This invariant helps generalize our code throughout the back end of the compiler, admittedly at the cost of a slight reduction in the run time performance of the generated program. As a result of this design decision, each sequence of operations or procedures which each depends on the result of a previous one, can easily chain together by simple popping the result of the previous one from the stack.

5.3.1 Assignment procedure

First, we calculate the right-hand expression, which is then pushed to the top of the stack. Afterwards, we pop this to the *RBX* register for temporary storage, the expression on the left-hand-side is now reduced to an identifier. We then move the right-hand side into the address corresponding to the calculated left-hand-side expression.

5.3.2 Arithmetic Operators

The main concern regarding arithmetic operators is first identifying whether or not an expression results in an integer- or floating point value. The expressions containing the operands are then popped from the stack. The given arithmetic operand is then applied and the result is pushed to the stack. If the operands are floating point values, we switch between the SSE registers and x87 floating-point instruction set.

5.3.3 Conditional Operators

Because we use x87 floating-point numbers, the standard comparison operators function as expected. When it comes to char values, we must take special care when comparisons are performed, as these strictly require the byte-level instruction set to work as intended.

To account for this, we check if the left-side of an expression gets evaluated to a value of type *char*, and then refer to the popped registers where the operands are stored in on the byte-level.

5.4 Control Structures

5.4.1 If Statements

The implementation of *if* statements is relatively simple. We place a jump statement at the start of the conditional label above the code of the body of the statement. If the conditional statement evaluates to true, we jump back up to the code of the body, once the code of the body is finished, or the conditional did not evaluate to be true, we jump to the end label of the conditional, which then exits the code for the if statement.

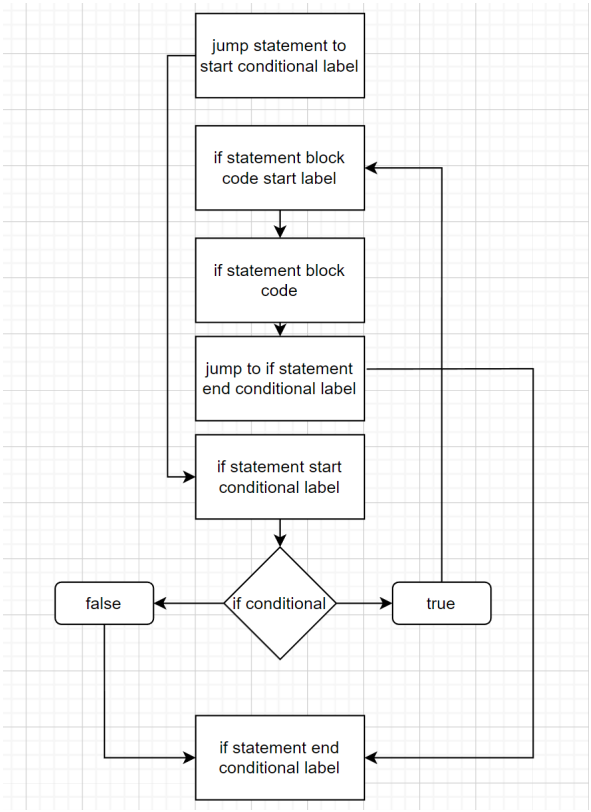

Fig. 23: Control flow of if-statements in Eridu

5.4.2 While Statements

We have implemented while statements in a similar fashion to that of our if-statements, as they share a relatively similar logic. The main difference comes, when, if we have entered the while statement's block code, and come to the end of it, we do not jump to the end of the conditional, but instead simply let the logic flow continue down back to the conditional statement logic once more.

5.5 Run Time Stack-Frame

The run time Stack-Frame consists of only a few dedicated registers and space for the variables to be used. In the following sections, we will briefly explain the layout of these stack frames.

5.5.1 Registers in The Stack-Frame

There are only two registers that are dedicated for the entirety of the run time, if we do not count the stack- and instruction-pointers. The first register is *RBP*, which is the base frame pointer. This is the register which points to the beginning of the current stack-Frame. The second register is *RDX*, which we use to keep track of our heap pointer. In section 5.7 we will go into further detail as to why we designed it this way.

5.5.2 Variables in The Stack-Frame

Variables are stored and referenced as an indirect-relative pointer to the stack frame. All variables are stored as (64-bit) quad-words. We chose to use this approach as it simplified some of the operations of the compiler. While modifying this could lead to potential speed-ups in the code during run time - since not as many inherent move operations are needed - it would also reduce our memory overhead. We feel that for the scope of this project, it was the more favourable approach to go with the static type size of 64 bits, as it allowed us to focus on some of the more intricate features of our language.

Pointers are created using a simple leaq instruction from the desired address (which would store a variable) and moving it into the offset from the base-frame-pointer containing the pointers symbol. To help visualize, the stack-frame, we provide figure 24 which can be found on the next page.

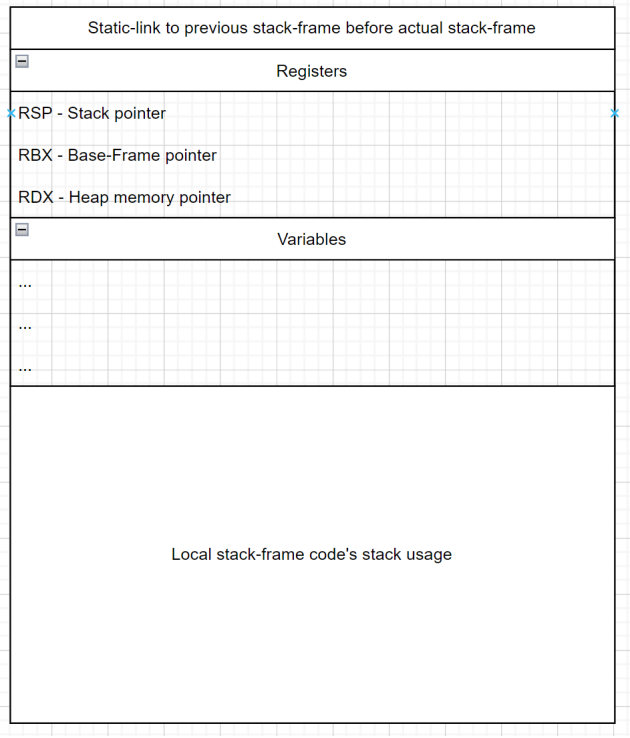

Fig. 24: Example of a stack frame in Eridu

5.6 Scoping & Blocks

During run time, different scopes are managed by different stack-frame structures. Upon entering a scope, we push *RBP* to save a reference to the previous stack-frame. This allows us to both restore the previous stack-frame upon leaving the scope, but also enables us to use this reference to the previous stack-frame as a static-link-chain which we can traverse. By doing this, we also remove the need to have a dedicated static-link register at all times.

After having pushed *RBP*, we set this register to the current value of stack-pointer. This makes it so that the current top of the stack is the start of the next frame. Upon leaving a scope, we free up all the stack space used by the frame by simply moving the address stored in *RBP* into the stack pointer. Since this was the beginning of the frame, we have essentially freed this stack-space for overwriting. Further, using an indirect-relative reference to quad-word before the beginning of the current frame, we can restore the previous base-frame-pointer. This is a reference to the earlier stack-frame, so we pop this value of the stack and into *RBP* once more.

5.6.1 Local Variables in Inner-Scopes

Local variables are stored in accordance with the procedure described earlier. The reason it works, and does not overwrite previous values at the same offsets from the base-frame-pointer, is simply because we changed which base-frame is referenced by *RBP* as described before. And thus, we are safe from overwriting variables of earlier scopes.

5.6.2 Static-Link Traversals

Using the address of the earlier base-frame-pointer stored before the current one, we can safely get values of variables declared in previous scopes. First, we store the current base-frame pointer in register *14*, so that we can restore the reference later. Then, we use the aforementioned earlier reference, and load this into *RBP*. The reason for doing this instead of using another register for static-link traversal, is because it simplifies the variable offset logic, if a variables address is always calculated as an offset from *RBP*. After locating the variable in the correct scope, we push it to the top of the stack, or in the case of an assignment, perform the assignment in the reached scope, before restoring the base-frame pointer from register 14. On figure 25 we show an example of this.

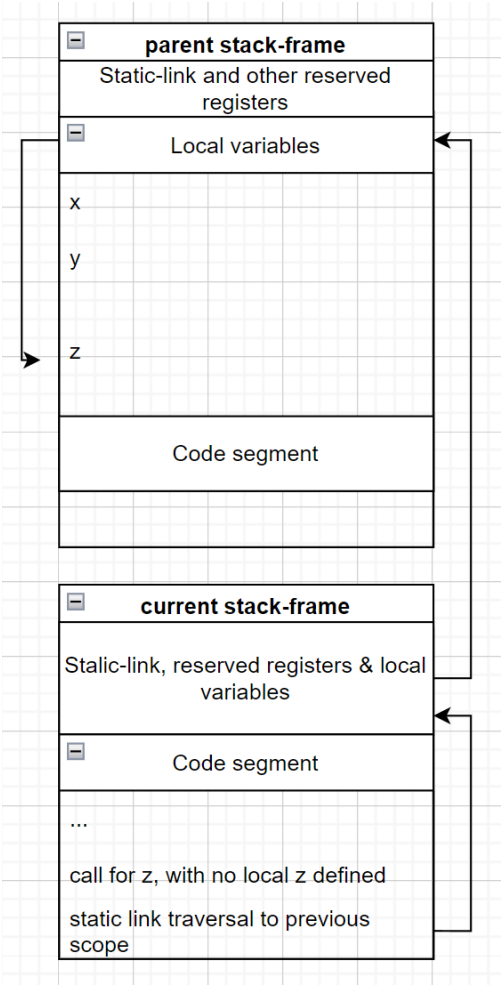

Fig. 25: Static-link traversal in Eridu

5.6.3 Functions

The implementation of functions is relatively similar to that off a block, with a few key differences. First, the code for functions is segregated to the bottom of the source file, separated by their respective starting labels. Once the function has been called, it operates similarly to the block-scope. The difference lies mainly in how we access parameters and the pushing of output to the stack before returning from the call.

Parameters are pushed to the stack going left to right, such that the first parameter in the parameter list is pushed first and then the next, operating like this until the final parameter has been pushed. A parameter to be passed can also be the result of an expression, without the need for much special care, making our stack-push invariant of operations and procedures.

Before calling the function through it's label, we push the base-frame pointer, which will serve as a place for the instruction pointer to return to once the function is finished and ret is called. Upon returning to first instruction after the call, we remove the parameters from the current stack and the base-frame pointer reference.

Inside a given function's code, we do not create an additional copy of the parameters. Instead, we refer to the copy pushed to the stack. We refer to them relative to the base-frame-pointer in the function's own scope, while accounting for the extra space taken by the base frame push both before the function call, and the one in the functions base-frame setup.

Once a function reaches a return statement, it will push the result of the return statement's expression as evaluated in the scope of the function. Using register *%r13* as a temporary placeholder register until the value is popped or ignored in the caller's scope.

5.7 Dynamic Memory

For this project, we chose to maintain a reference to the heap ourselves. We do this by using the C run time binary version of malloc, and allocating a huge chunk of memory. This memory is the entire heap of the compiler in the current scope of the project.

We chose this approach, as we wanted to manage the heap space our program uses ourselves. As of now, there is no way to increase, or free memory once allocated during run time. The obvious next steps for our heap memory management would of course be a free command, which could be implemented using the Tombstoning technique, and increased storage space could easily be achieved by a second call to malloc and a transfer-by-copy to this new increased space.

As mentioned earlier, we keep track of the current allocated heap space in *RDX*. The program handles allocation request by first pushing *RDX*, which will serve as a pointer to the start of the heap memory requested, and then by incrementing RDX by the amount of space requested.

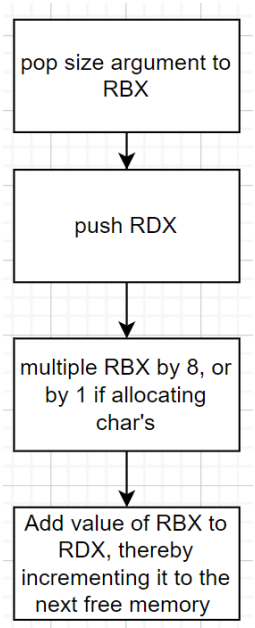

Fig. 26: Heap memory allocation in Eridu

The amount of space requested can be run time expression, so we first calculate this size, push it to the stack (Due to the nature of our stack-push invariant) and pop it into a register so it can used to offset *RDX*. In general, the process for allocating a block of memory can be seen in figure 26. From this example, the only exception to the typical type size of quad words can be seen, as to be compatible with the type of constant strings we employ, we must store individual chars on the heap as a single byte.

5.8 Callables

Callables are stored as pointers on the stack. For assignment of a callable, we store the first instruction of the assigned function in the callables offset from the base-frame-pointer by using the leaq command on the label of the function. This stores the first instruction of the function as a pointer in the callable.

When a callable is called as a function, we use indirect function call Assembly syntax to call the value stored in the callable. So for a given callable stored at say *-8(%rbp)*, we would call it using *callq *-8(%rbp)*. Parameter passing works as for normal function calls. An example of such an assignment and usage can be seen in the generated x86 assembly for the singleton example seen in figure 12.

```
# Save pointer to this stack frame before following static link
movq %rbp , %r14

movq 8(%rbp) , %rbp  # Performing static link computation

# Moving address of first instruction of function into temp register
leaq construct_logger_0(%rip) , %r13

# Push pointer value to top of the stack
pushq %r13

# Restore pointer to this stack frame after following static link
movq %r14 , %rbp

popq %rax             # Pop value from right hand side to rax

movq -8(%rbp) , %r13 # Move struct pointer to temp register

movq %rax , 16(%r13) # Move the value from reg 1 into struct member
```

Fig. 27: Assignment of a callable in generated Assembly

On figure 27 we show the assignment of a function from a different scope into a callable variable located in a structure, and how this procedure looks once it is compiled into x86 AT&T assembly.

5.9 Composite Types

The two compiler-level composite types provided by Eridu is the Array and standard C-like data structures. Both of these concepts only live on the heap, and the only reference to them is through a pointer stored on the stack.

5.9.1 Arrays

Arrays exist as a contiguous set of memory addresses in direct proximity to each other. Since there is no segmentation in our compound types once allocated, a reference to an array index exists as indirect-relative computation from the first index. Since the index can be a run time expression, we calculate it in the same way as we calculated the amount of space to allocate during allocation of heap space. We then add this calculated value to a copy of the arrays pointer, which will increment it to the desired index. An example of this, along with a Eridu code representation of the situation can be seen in figures 28 and 29. The line in figure 28 corresponding to the array-indexing flow in figure 29 is the line *print(arr[2])*.

```
int * arr = allocate(int, 3);

int i = 0;
while (i < 3) {
    arr[i] = i + 1;
    i = i + 1;
};

print(arr[2]); // prints 3
```

Fig. 28: Example using an array index

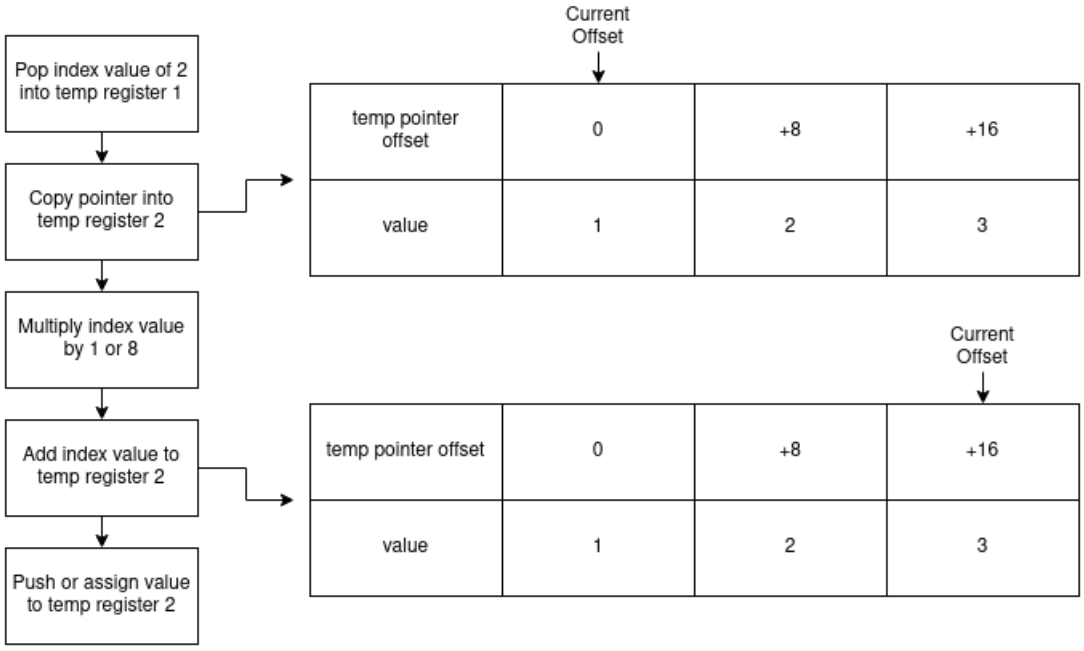

Fig. 29: Array Indexing Control Flow in the Code Generation Phase

5.9.2 Data Structures

The way data structures are stored on the heap is identical to that of arrays. The main difference is that it is decide able at compile time, which structure member is referenced. Therefore, the calculations to reach this index is reduced.

6 Testing

6.1 Gula - Testing Framework

As for any large project, mitigating compile and run time errors is important, and as such, we wanted to come up with a testing framework which could help us locate and detect errors with regards to not only the compilation and execution process, but also verify the expected outcome of logical procedures. This framework should allow us to both test that our compiler implementation was correct, but also allow developers using our language to ensure their code generates the results they expect. Furthermore, we wanted to avoid the separation of source files and their expected outcome, as making it fully self-contained can help organize and maintain test suites.

Our solution to this was to develop a syntax that can be embedded as comments, inside a given Eridu source file. This allows developers to be close to the code they want to test and quickly evaluate and modify their expected outcomes, while still being able to compile and execute source files without interference from said syntax.
As seen in figure 30, we encapsulate the expected outcome definitions using start- and end markers, as to differentiate them from regular in-line comments. The placement of this section has no impact on the testing process, which gives developers room to position it as they like.

Each in-line comment inside these sections represent some value (type-invariant) sent to *stdout*, which in Eridu, is done by using the builtin *print* function. However, some values, like memory addresses, can not be known before run time, sometimes you might even expect your program to result in an error. As a way to solve issues like these, we introduce the concept of macros.

```
//%% OUTPUT
//value1
//value2
//value3
//%% END
```

Fig. 30: Snippet of in-code expected outcome definitions

The problem regarding memory addresses most often occur when a developer is trying to assert whether or not two or more addresses are the same. The macro *$$X* (where X is an integer), denotes a variable whose value is bound by the output at the line it was first used in. These variables can then be used to express that a value on another line is expected to be equivalent, as illustrated in figure 31.

Similarly we also implemented *$$segfault* and *$$error* macros, as testing programs known to be faulty can help with the overall coverage of test suites. To showcase how the framework and the expected outcome concept works in practice, we refer to figure 32, which shows the use of expected outcome to assert the correctness of a simplification of the iterator pattern shown in chapter 6.7.

```
define int main() {
    // Create dummy data
    str * names = allocate(str, 5);
    names[0] = "Michael";
    names[1] = "Peter";
    names[2] = "Hamurabe";
    names[3] = "Jackson";
    names[4] = "Wolfeschlegelsteinhausenbergerdorff";

    // Default iterator is a linear iterator
    struct Iterator * iter = create_iterator(names, 5);

    // Unpack .next callable from struct
    callable next = iter.next;

    str a = next(iter);
    print(a);
    a = next(iter);
    print(a);

    return 0;
};
main();

//%% OUTPUT
//Michael
//Peter
//%% END
```

Fig. 32: Snippet of expected outcome used to test an iterator

As we prioritized ease-of-use with regards to the expected outcome definitions, we wanted the framework itself to be easily interpretable and easy to use, this is exemplified by the small amount of configuration options, which we expanded with features in an as needed fashion. On figure 33 we showcase the user-facing options menu, as seen when called with the *'-help'* flag.

```
//%% OUTPUT
//Testing Gula Macros
//$$1
//$$1
//They're the same
//%% END
```

Fig. 31: Macro usage in expected outcome definitions

```
Gula - Testing Framework for .erd source files.

Usage: gula.py [-verbose] -path <folder/filepath
 -src <folder containing ./compiler> [-step]

            -step: Pauses after each test.
            -verbose: Prints the output of the compiler.
            -help: Prints this help message.
```

Fig. 33: Output of the '-help' flag

Our testing framework was implemented using Python, as our requirements favored ease-of-development, reasoned by both the projects time constraints, and within reason, not being sensitive to execution speed. Lastly, we show an example of the framework being used to test source files in a dummy environment, and how the data is presented to the user.

```
$ python3 gula.py -src ../dev/src -path meta -verbose

Verbose mode enabled
Beginning Testing Framework on Folder: 'meta'
Recursively searching for .erd files.
Test Passed: 'meta/test_gula.erd'
Test Failed: 'meta/test_gula2.erd'
        Expected #1 Output: '42'
        Actual #1 Output: '24'
Test Passed: 'meta/test_gula3.erd'
Test Failed: 'meta/test_gula4.erd'
        Expected #0 Output: 'Foo'
        Actual #0 Output: 'Bar'


2/4 Tests Passed
50% of tests passed
```

Fig. 34: Output of a test suite execution

As a final remark, it should be mentioned that this type of testing was not always sufficient, as the underlying reasoning behind a given problem did not get analysed or inspected by the framework, and thus tooling such as the memory debugging tool *valgrind*, and the *GNU Debugger* were a vital part of the development process.

6.2 Compiler Design & Architecture

Developing complex systems usually calls for a lengthy period of design and planning before implementation begins. This stage typically consists of diagramming, researching design patterns and writing a formal description of both the requirements of a system, and their respective priorities.

The architecture of a compiler often follows a general structure wherein responsibilities are delegated into smaller segments, and data is passed in a unidirectional fashion through a each layer. In our project, we decided to adhere to this practice, as efficient and maintainable compiler architecture is often something learned by an iterative process of reflection and refactoring, which the scope of this project did not allow for. [1, p. 4] The architecture of the Eridu compiler as shown on figure 35 shows this unidirectional pipeline, starting at lexical analysis and ending in the compilation of our intermediate representation into x86 AT&T assembly.

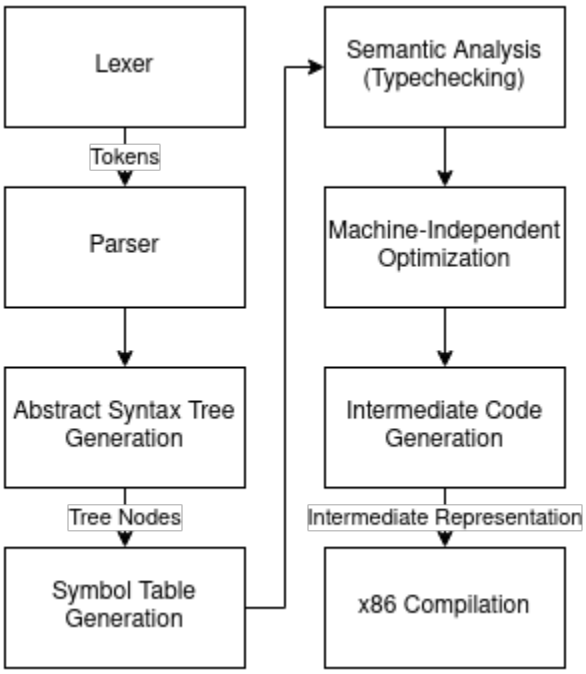

Fig. 35: Architecture of the Eridu Compiler

This architecture allows us to bypass phases of the compiler during testing and development, making it possible to turn off phases such as optimization or type checking with the press of a button, and further allows us to easily plug-and-play other compilers which take our intermediate representation and converts it into other architectures such as arm. Looking more specifically at the internals of the compiler, we tried to implement software engineering principles such as loose-coupling, single responsibility and modular programming. Admittedly, with complex systems these patterns can become hard to preserve, and with the need for refactors or bug-fixes, was not strictly adhered to, but rather used as an advisory guide during development.

6.3 Style guide & Formatting

While the aesthetics of a codebase is generally not regarded as a vital factor, it is proven that even advisory coding standards such as naming- and documentation conventions can make code more readable and in fact, reduce the probability of writing faulty code and facilitate better communication between developers [5].

Our first steps to integrate a style guide in our project, was to set strict conventions for both naming, styling, and logging information, allowing us to quickly understand the identity of object or an error. We then combined our style guide with the *clang-format* formatter, which enabled us to set a de facto standard for how code should look, which in turn, allowed us to create consistency and predictability throughout our project.

7 Implementation Notes

7.1 Custom Intermediate Representation vs. LLVM

Often, modern compilers would use a tool chain in the code generation and emit phases of the compiler. LLVM or one of it's derivatives is famously a popular choice for these steps, and due to this we considered the positives this might have given us.
However, upon weighing all the positives and negatives of having a custom representation versus that of LLVM, we decided to implement our own custom intermediate representation.

We chose to do this, as we felt it would be more aligned with the spirit of this project, which we consider to be a great opportunity for learning intricate implementation steps of a typical compiler. In this process we not only greatly improved our understand of the x86 assembly language and programming skills; we also gained an insight into how compilers generalizes common high-level language features, so that they can be used as a template to complete tasks in a generalized fashion.

The small negative to this approach is the speed of development. We expect that an increase in code output could have been achieved by using the LLVM tool chain, but taking the aforementioned positives of custom representation into consideration, we felt that we were justified in our chosen approach.

7.2 Linear Search vs. Hash Table Optimizations

During the development of the project, we had many learning experiences, some of which gave us things that could have been implemented better in hindsight. One of these is the data structure of our symbol tables.

The implementation of this in Eridu is a linked-list (stored as a pointer-pointer of symbols), which gives us a linear search time for looking up symbols in a given symbol table. Since symbols are added to the list in order of declaration, their offsets will also be in order of declaration. This comes into play once the code generation phase is considered. In this step of the compiler pipeline, we need to know the order of declaration to give the expected variable offsets for our stack-frame.
While it would be fine to have seemingly quasi-random offsets for a local stack-frame, this factor

becomes more significant in the case of parameter passing. Because parameters are pushed left to right, the corresponding offset in a given function's own stack-frame would be potentially jumbled, because the expected offset (as it is calculated from the position in the symbol table) could be misleading.

This was our reasoning for using a linked-list type representation in the symbol tables. However, as we got more experienced with compiler architecture and design, we realised that a hash-table implementation of the symbol tables would have been possible, had we stored the declaration offset along with a given symbol and used this offset for the code generation stage, instead of simply counting a symbol's position in the symbol table.

This would have given us an average case constant search time for symbols, which is an obvious improvement over the standing linear search time. An easy improvement that would have been a next step would have been to refactor the symbol table data structure as described.

7.3 Constant Folding

One of the more common compiler optimization techniques used by many modern compilers is constant folding [4, Ch. 12.1, p.29]. Constant folding refers to the identification and subsequent evaluation of expressions which consist purely of constant values. This evaluation can be performed at compile time, rather than run time, reducing the number of instructions needed to be executed whilst running a program. In this case, constants and constant expressions refer to literal values such as integers and floating points, and does not extend to variables at the current stage of development.

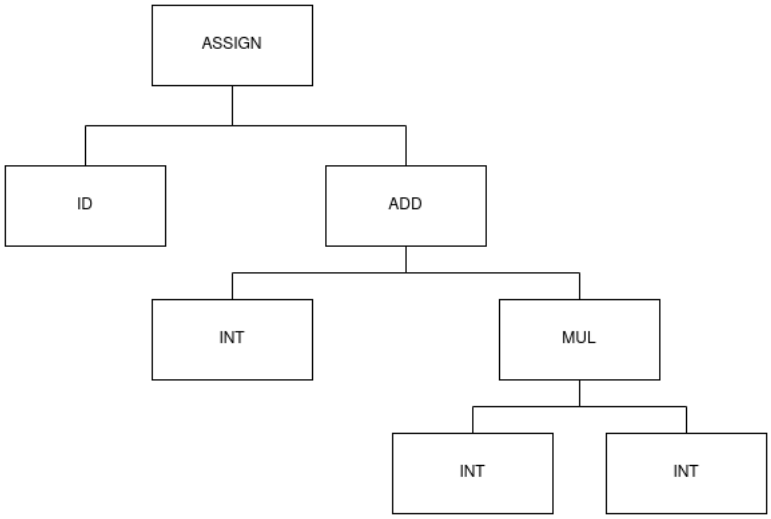

Fig. 36: Abstract Syntax Tree for the expression *3 + 2 * 5*

When performing constant folding, we enter a space where very explicit and well defined design decision has to take place, as developers need to be able to clearly and intuitively understand what

these implicit optimizations actually do. A simplified example of a flawed system could be constant folding whose order-of-operations (arithmetic precedence) does not coincide with the order instructions would be executed during run time.

We tried to mitigate this potential problem by performing constant folding in a bottoms-up fashion, traversing to the bottom of a given sub-tree which is constructed using the precedence rules specified in the parser, and folding upwards.

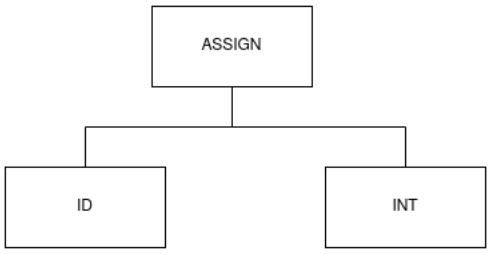

Fig. 37: Abstract Syntax Tree from figure 36 after Constant Folding, INT is now *13*.

On figure 36 we show an example of a sub-tree representing the expression *3 + 2 * 5*, as multiplication has a higher level of precedence than addition, this would be folded first using bottoms-up traversal, maintaining the proper order-of-operations. The resulting sub-tree can be seen on figure 37. Related optimizations such as constant propagation and common sub-expression elimination were implemented in the Eridu compiler, however, as we did not have time to vigorously test these optimizations we felt that removing them from the build version of the compiler was the more responsible choice, and simply mention that we are working on these optimizations in a future iteration of the compiler.

8 Performance Evaluation

Following the design, implementation, and testing of the Eridu compiler, we now want to measure its performance against other languages to gauge the usability of our language from a different point of view than the developer experience. In the following sections we will go into detail about how we performed our bench marks, and try to both interpret and discuss the results in a meaningful way.

8.1 Utu - Bench marking Framework

In the same spirit as for our testing, we wanted to implement our own bench marking framework which would allow us to easily plug-and-play support for different languages and programs. We named this tool *Utu*, which is a Python program that uses the *timeit*[4] module from the Python standard library.

As shown in figure 38, adding new bench marking programs can be done by adding a sub-folder to the *benchmarks* folder, each of which should contain the source code for a given program for each language bench marks should be

```
utu.py
benchmarks/
    /knapsack
        knapsack.c
        knapsack.cs
        knapsack.cpp
        knapsack.erd
    /fibonacci
        ...
    /bubblesort
        ...
```

Fig. 38: Snippet of the Folder Structure used by Utu

[4]https://docs.python.org/3/library/timeit.html

performed on. Adding support for a new language inside Utu is trivial, as it utilizes Python meta programming to automatically detect whether or not a given file type has a corresponding test function. Using the Python *subprocess*[5] library, each tester first performs the required setup for a source file. Looking at figure 39, an example of both the setup, tear down and execution of C and C source files can be seen. Implementing support for a new language is a simple matter of writing the relevant setup and tear down logic.

```
def test_c(path, src):
    """ Test a C file. """
    subprocess.call(["gcc","-O3","-static","-o", "test", path])
    time = timeit(stmt="subprocess.call('./test',
                  stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)",
                  setup="import subprocess", number=number_of_runs)

    subprocess.run(["rm", "test"])
    return time

def test_cs(path, str):
    """ Test C# File """

    subprocess.call(["mcs","-optimize","-out:out.exe", path])
    time = timeit(stmt="subprocess.call(['mono', 'out.exe'],
                  stdout=subprocess.DEVNULL,
                  stderr=subprocess.DEVNULL)",
                  setup="import subprocess", number=number_of_runs)

    subprocess.run(["rm", "out.exe"])
    return time
```

Fig. 39: Example of Testing Functions in Utu

The *timeit* module by default measures the *total* run time of the given number of runs in seconds. The average time per execution of a given program can then be calculated by simply dividing the resulting times by the number of runs. By default, *timeit* uses a timing method called *perf_counter*, which we changed to use the *process_time* method, this returns the sum of the system and user CPU time, which we did to ensure that any differences in how these languages use interact with the operating system would also be measured. Having now implemented a framework with which we could add and run new bench marking programs, and easily store these results, we began the process of deciding which algorithms we should use for bench marking.

[5]https://docs.python.org/3/library/subprocess.html

8.2 Bench marking

The first choice we had to make regarding our bench marking, was which languages we felt would give the best insight into different performance metrics. Given our inspiration from C, we decided that comparing Eridu against different languages in the C family would be the most relevant, as both their intended use and core philosophies are more in line with the primary objectives of our language. We furthermore wanted to include both imperative and object-oriented languages, as we in section 2.10 showcased how our language can be used to emulate some of the features seen in this paradigm. From here, we will briefly summarize the core ideas of each of the languages we chose to bench mark against:

- **C** - This was an obvious choice, as this is the language to which we have compared ourselves to with regards to programming style, feature sets, type systems and so on. C is as should be evident, a C-like language, which like Eridu also utilizes focuses on a purely imperative paradigm.
- **C++** - Following in the footsteps of C, C++ is in spirit an extension to C, adding support for classes and opening up the object-oriented programming paradigm to developers. We felt that this was a good language to bench mark against, as it like C is a language which has had a long time to evolve and added many of the functionalities that we wanted to be able to approximate, as core functionalities.

There are some specific compilation properties for each of these languages, which should first be discussed, as to assert the exact run time and compilation configurations which we chose. For both C and C++ we compiled them using gcc and g++ respectively, both with version 9.4.0, static-linking enabled and with the optimization flag set to O3. In all cases, we chose to run Eridu against these languages in an optimized compilation, as we felt this would most realistically compare our run times to those of the others.

For bench-marking, we wanted to chose tests which covered different aspects and use-cases of the compiler and see how we compared to the main C-family languages. These tests were:

- Knapsack problem: we chose to test with the knapsack problem, as this is a famous NP-complete problem.
- Bubble-sort: we chose bubble sort, as this sorting solution has a notoriously slow worst-case, which we found fitting for a bench-marking scenario, as this puts more "stress" on the language at run time compared to other, faster algorithms.
- Hello-World: While this test is very simple, it tests the average speed at which a language can write to standard output.
- Fibonacci: We chose this problem, as it is a simple recursive formula with relatively few operations, which allows us to test specifically for the speed of recursions compared to the other languages.
- Naive pattern matching: Finding a pattern in a string is common problem in many languages, with the naive pattern matching algorithm having one of the greatest number of comparisons performed, we chose this for the same reasons as we chose Bubble-sort.

- Finding Prime Factors: We wanted to include a more mathematical problem to the list of bench-marks, to test the speed of our arithmetic operations in relation to the C-family.

We chose to perform the tests in sets of powers of tens (with the same given input), up to 10000 runs and then average the result for each language in a given set. The reason we did this is that we wanted to check how we performed on average over many different runs to account for variations in individual runs. Additionally, we performed the same tests with different scales of complexity . We felt in turn this gave a better perspective on the scale ability of Eridu's performance.

8.3 Hardware

As to ensure the validity of the data, we made sure to perform all tests on the same machine at similar states with regards to other running processes and computations. Further, we note that the platform used to perform the bench marks ran the Ubuntu 20.04.4 LTS operating system. Before we begin discussing the results of our bench marking, we here show a detailed description of the hardware featured on the bench marking platform:

RAM	8GB of 3600 MHz DDR4 SDRAM
Motherboard	Asus B550m-E with the AMD B550 Chipset

CPU	# Cores	Clock Speed	# Threads	L2 Cache	L3 Cache
AMD Ryzen 7 5800X	8	3.8GHz	16	4MB	32MB

GPU	Core Clock	Memory Clock	Memory Size	Memory Type	Memory Bandwidth
GeForce RTX™ 3060 Ti	1740MHz	14000MHz	8GB	GDDR6	448GB/s

8.4 Results

We provide here the results of our bench-marking, while trying to explain some of the results along the way. The first set of data we want to showcase, is the average times over all tests, for all languages. Figure 40 represents the average times based on 10.000 runs, and shows that Eridu while not quite as fast as C or C++, manages to do all right for most of the tests. It becomes clear when looking at especially the knapsack and Fibonacci tests, that Eridu might not be as consistent with regards to efficiency across different types of computations.

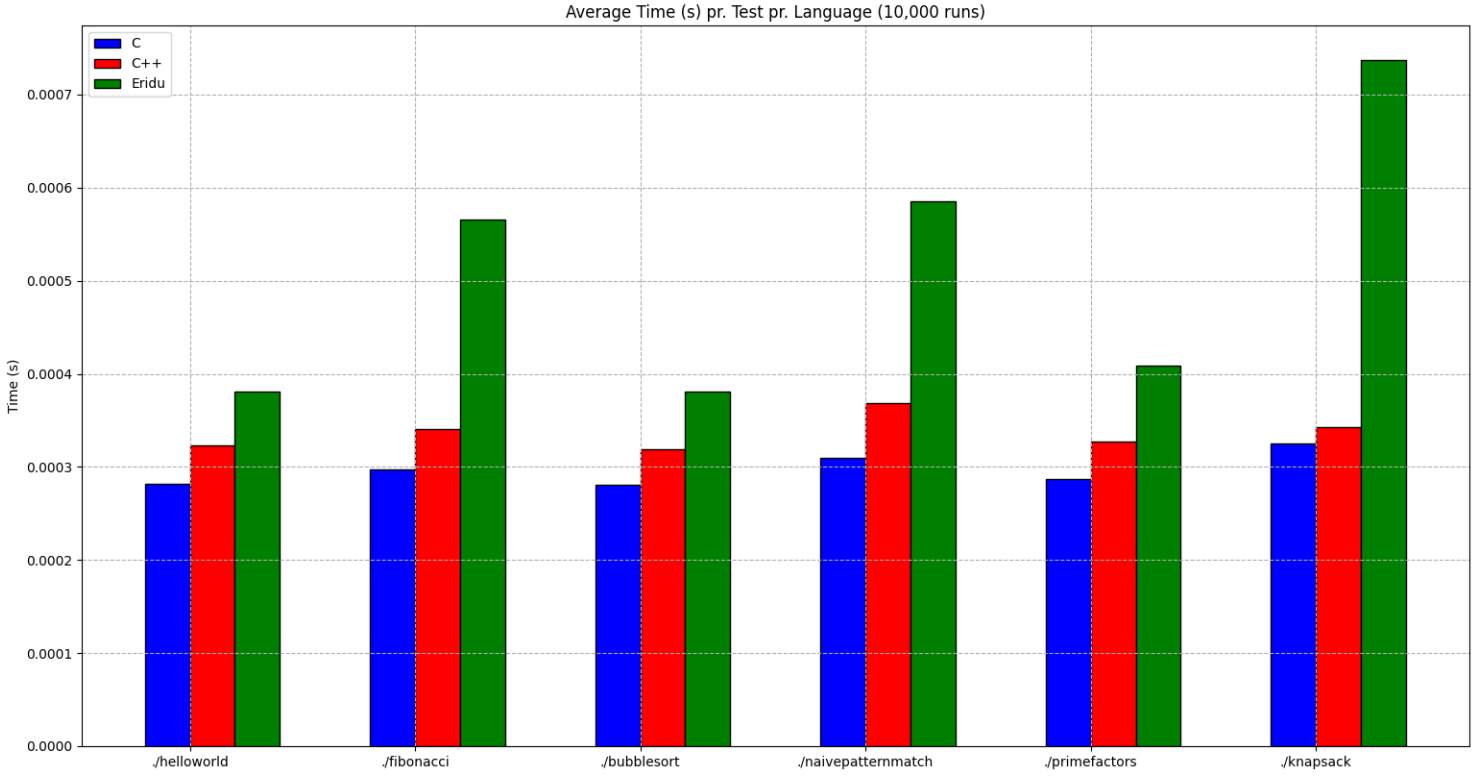

Fig. 40: Total Average Times for All Tests for All Languages

Before running our bench marks, we did in fact expect Eridu to perform slower than C and C++ in handling recursions, which is confirmed in the Fibonacci test. In general, we expect this slowdown to occur as a result of the base frame setup and tear down, as we use 2 extra instruction compared to unoptimized C and C++ for both of these procedures, which comes as a result of our static link storage method as was explained in section 5.6

Looking past the Fibonacci example, we also see a large deviation whilst testing the knapsack problem. This problem tests many aspects of our compiler, and it would be hard to pin-point a singular procedure where we expect a slow down. However, each of these aspect are explored in the other bench marks and can be more meaningfully explained in those portions. The aspects where we would expect slowdowns are array indexing, arithmetic operations, program setup and tear downs, as well as recursions. In addition to this, in the performed tests for the languages other than Eridu, the code examples use stack-allocated arrays where we instead use heap-allocated arrays, because of the implementation details explained in section 2.7.1. We did this because this way of declaring the arrays in the other language was the simplest way to implement the given problem. In this way, we also compare the speed of the "simplest" code setup in the languages.

This slow down which occurs during array indexing would also explain the slowdown as seen in the naive pattern matching example. Interestingly enough, we do not see as large a deviation in bubble sort, but this test does not perform as many array indexing procedures, which would explain the difference. Considering the cases where we perform worse, we felt it important to discuss the general scale-ability of our language when it comes to increasing complexity of programs.

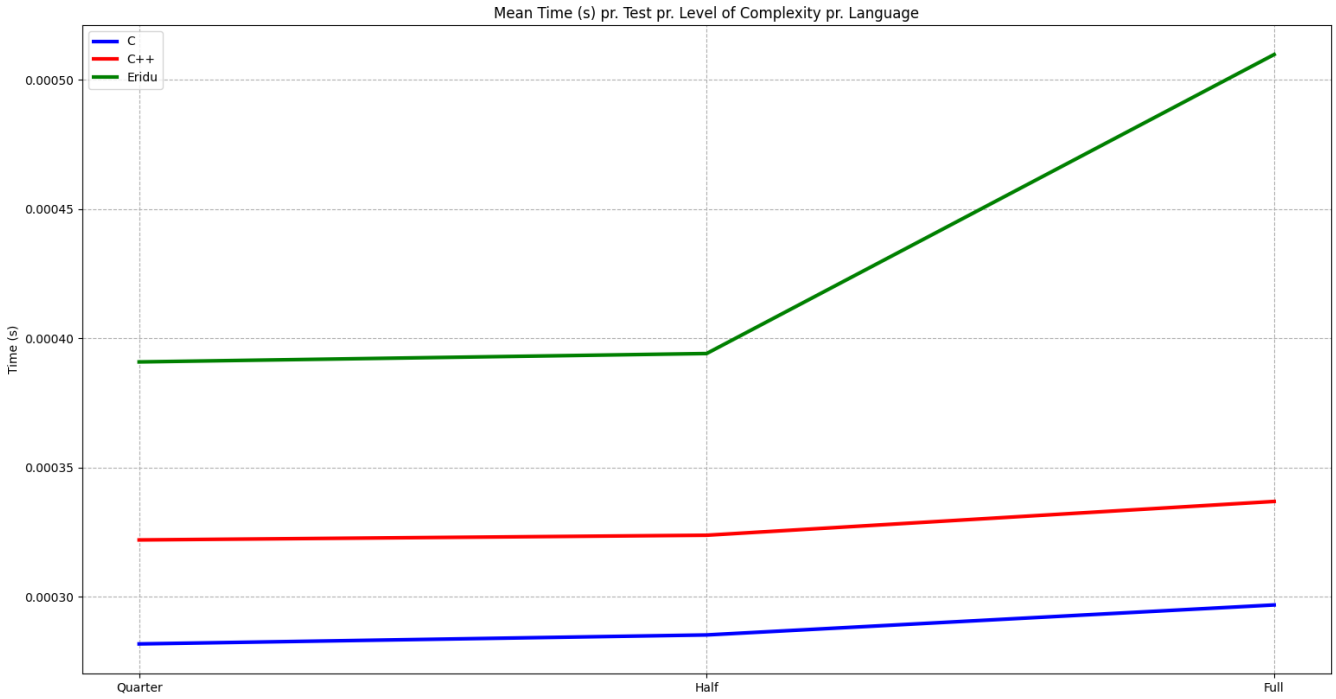

Fig. 41: Mean Times pr. Test pr. Level of Complexity pr. Language

As a way to measure this, we created a total of three variants of our bench marking programs, these three levels - quarter, half, and full - are designed as to increase in complexity, but as some problems can not simply be reduced by half, the actual differences are not necessarily precise. This however is not that important, as we are interested in looking at the relative increases in run time as compared to both C and C++. On figure 41, we see that the full tests, which we used for the previous bench marks, is considerably worse than its simpler variants. We attribute this to those procedures of ours which require those few instructions more.

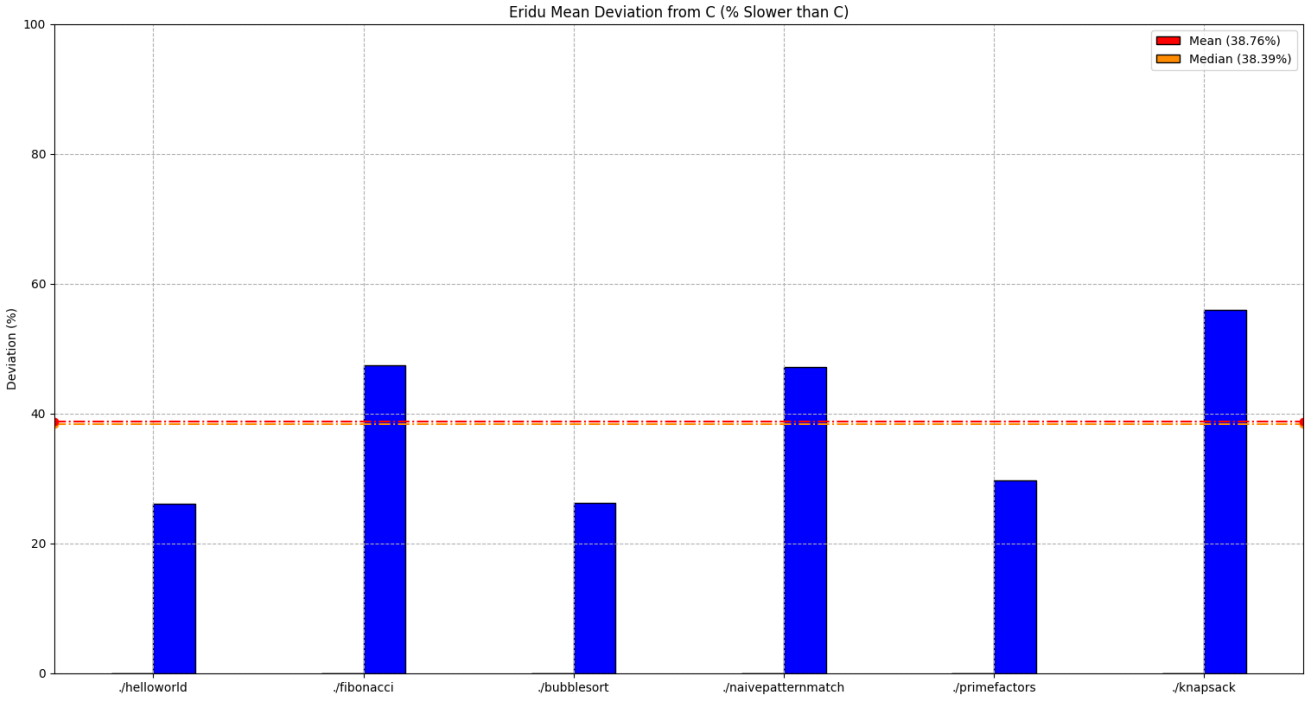

Fig. 42: Mean Derivation from C

The scale ability of Eridu becomes particularly visible when we look at the exact numbers which represent our run time compared to that of C. On figure 42, we see that our performance is about *38.5%* worse than C. However, as we discussed regarding the scale ability of Eridu, this data does not accurately represent the real life differences, as the average complexity of a computing problem which developers might face during a project can not be known. Because of this, we also show our mean derivation based on the bench marks which we called *half-complexity* on figure 43. Looking at this figure, we see than Eridu is about *27%* slower than C.

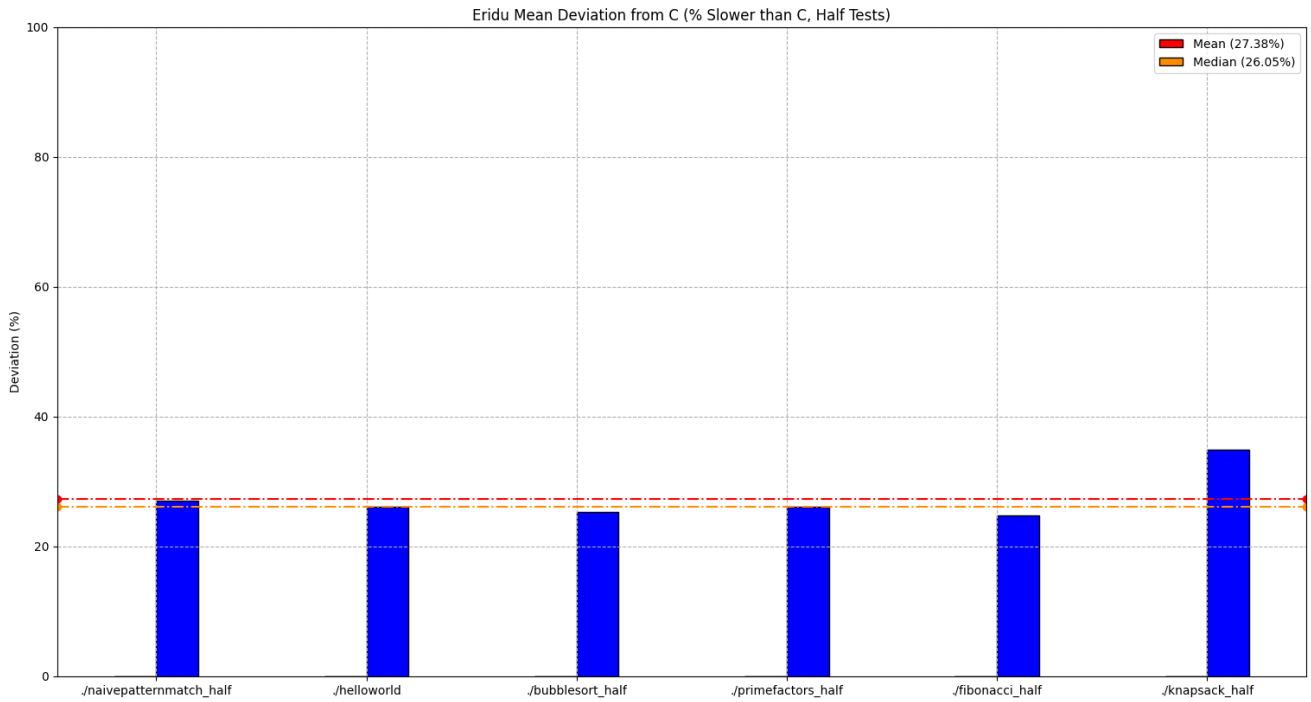

Fig. 43: Mean Derivation from C (Half Tests)

The reader might question why many of these tests take approximately the same time, and why the different computational tasks do not seem to be that difference when it comes to their run time. This can be attributed by the percentage of execution time which the program setup and overhead takes up. We did not want to normalize this out of the data, as we felt that the overhead of a given programming language and its execution environment was also an important factor. As to give the reader the opportunity to compare these results without their respective overheads, we on figure 44 show the average time which an empty program took to run. These programs represent the minimum amount of code which is required to start running code in each language.

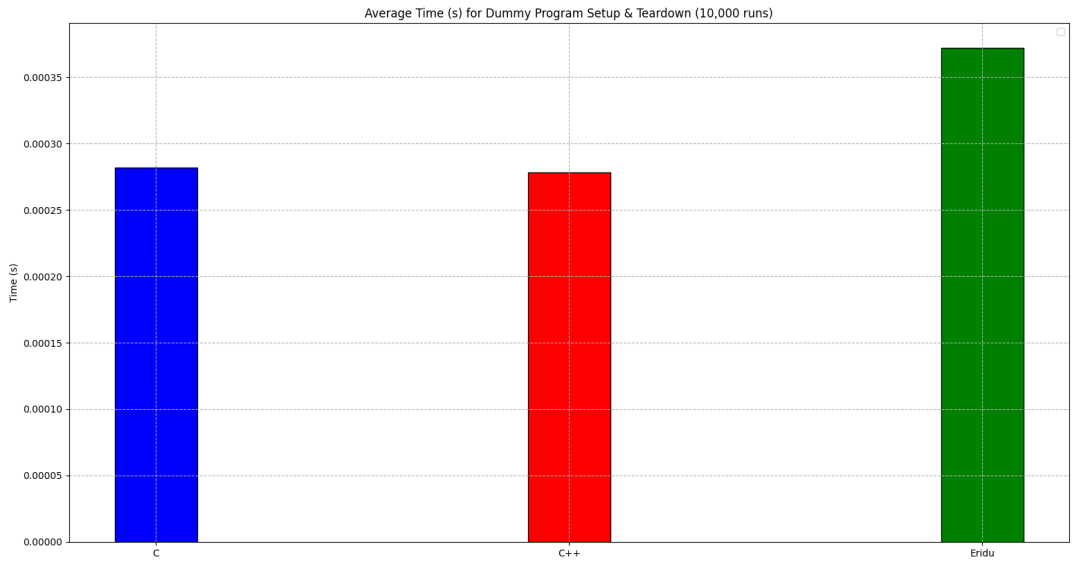

Fig. 44: Average Time for Dummy Program Setup & Teardown

Before we proceed, we refer the reader to figures 46 and 47, where the raw data used for these tests can be found. Further, a *JSON* is included in the attached files for this thesis. As our final remarks with regards to bench marking and our results therefrom, we till briefly mention a few other considerations which we have made throughout this process.

First we want to note that a potential contributor to the slow downs seen in Eridu are assignments. This procedure uses the stack to push and pop values for assignments, instead of assigning directly to whichever variable or index is being assigned to, the reason for which is described in section 5.3. Additionally, we note that we do not expect these results to perfectly represent the efficiency of our language, as there are many more computational problems which we have not performed, each of which might play to our strength or weaknesses.

To quickly touch on the prime factors benchmark, we did expect to be slower than C and C++. Comparing the amount of instructions we use per arithmetic operation with unoptimized C, to our knowledge we do not use extra instructions for the basic operations, however, the efficiency of our home-brewed square-root function as compared to that of C and C++ might not be equivalent, which could account for some percentage of the slow down seen. In addition to prime factors, we also bench marked using a simple *hello world*[6] program. This bench mark primarily tests the output speed of Eridu. Since Eridu's print statement's are one of the more inefficient aspects of the compiler, we expect to be a lot slower than the other languages here. Depending on the expression being printed, we are have around 8-9 extra instructions compared to unoptimized C.

One of the important steps of the bench marking process which we feel should be discussed, is the manner in which we ensured that all of the algorithms we used were implemented in fair and properly comparable ways. On the website geeksforgeeks.org, common solutions to all of the problems which we used can be found. For each of our bench marking programs we copied the implementations from this website and tried to implement a version in Eridu which worked as similarly as possible. This is also a factor when it comes to the difference in speeds seen across our bench marks, as solutions tailored to Eridu's strengths could perform differently.

9 Concluding Remarks

Throughout this project, we have come to understand the impact which compiler design and development has on the landscape of modern programming. The power of abstractions and the ability to implement complex logic at a higher-level, where writing machine specific assembly instructions is simply not a viable approach. Further this project has given us invaluable insight into the inner workings of programming languages, as well as the different paradigms therein, which in turn has made us better developers.

For this project, we sought to implementing a wide amount of features instead of creating a smaller set of features which we would polish to perfection. This allowed us to explore the inner workings of more advanced concepts such as callables, and in our opinion, increased the general usability of the language. Our goal was to write a language which we would find fun to use, and which lives up to our ideals as mentioned in 1.1. While mostly using the same syntax as other languages in the C-family, we feel that the small changes we made - such as very explicit naming of the dereference intrinsic - can positively impact programming projects.

[6] https://en.wikipedia.org/wiki/"Hello,_World!"_program

In the future, we would like to further generalize some of the implementations in our compiler, especially the current need to "unpack" composite types stored inside other composite types. Along with this, we also want to take the unfinished compiler optimizations that did not make it into our project, and make them ready for a proper build version. The grammar of Eridu could also use some polishing up, which would be one of the first steps we would take were we to continue development. And if time permits, we would also like to implement an assortment of other language features such as list comprehensions as seen in Python[7], and extend support for functional programming using partial evaluation and currying techniques.

In section 3 we discussed the implications of not using Bison in our tool chain. For future iterations of Eridu, we would like to empower this stage, and make it responsible for some of the basic functionality seen in our symbol generation and type checking. This is not to say that we want to completely merge the functionality of the different stages as they are now, but rather reduce the number of tree traversals we have to perform.

In conclusion, we can say that when it comes to our final product, we are relatively satisfied with the outcome. While this project at first seemed daunting, we ended up with a product which we are proud of, and with each step in the development process, we came to appreciate the field of compiler development even more.

References

[1] A. W. Appel and M. Ginsburg. *Modern Compiler Implementation in C.* Cambridge University Press, USA, 2004.

[2] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1 edition, 1994.

[3] K. S. Larsen. Scil - *A Simple Compiler in a Learning Environment.*

[4] S. S. Muchnick. *Advanced compiler design and implementation.* 1997.

[5] S. Popić, G. Velikic, H. Jaroslav, Z. Pavkovic, and M. Vulić. The benefits of the coding standards enforcement and its impact on the developers coding behaviour-a case study on two small projects. 11 2018.

[6] D. Ritchie and B. Kernighan. *The C programming language.* Bell Laboratories, 1988.

[7] https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions

```
// Vehicle Container
struct Vehicle = {
    str name,
    <int> callable <struct Vehicle * > simulate,
};

// Vehicle Specific Simulators
define int _simulate_t45_model_90(struct Vehicle * t45) {...};
define int _ariane(struct Vehicle * ariane) {...};

// Vehicle Constructors
define struct Vehicle * make_t45_model_90() {...};
define struct Vehicle * make_ariane_srb_delta2() {...};

define int simulate(struct Vehicle ** vehicles, int size) {
    int idx = 0;
    callable sim;
    struct Vehicle * vehicle;
    while(idx < size) {
        vehicle = vehicles[idx];
        sim = vehicle.simulate;

        int result = sim(vehicle);
        if(result != 1) {
            return 0;
        };
        idx = idx + 1;
    };
    return 1;
};

define int main() {
    int num_vehicles = 2;
    struct Vehicle ** vehicles
        = allocate(struct Vehicle *, num_vehicles);

    struct Vehicle * t45 = make_t45_model_90();
    struct Vehicle * ariane = make_ariane_srb_delta2();

    vehicles[0] = t45;
    vehicles[1] = ariane;

    if(simulate(vehicles, num_vehicles) == 1) {
        print("All vehicles simulated successfully");
        return 1;
    };

    print("A vehicle didn't work");
    return 0;
};
main();
```

Fig. 45: Generic Vehicle Simulator

./program_setup_teardown	
.erd	0.0003722284444499342
.cpp	0.00027820738630034614
.c	0.0002819883536998532
./helloworld	
.erd	0.00038080860039990514
.cpp	0.0003228221348996158
.c	0.0002816020368001773
./fibonacci	
.erd	0.000565416499300045
.cpp	0.0003410260101001768
.c	0.0002971040976997756
./bubblesort	
.erd	0.00038067320890040716
.cpp	0.00031872601290015157
.c	0.00028052886369987393
./naivepatternmatch	
.cpp	0.00036835270209994634
.erd	0.0005854264251996938
.c	0.0003095355213001312
./primefactors	
.c	0.0002873329427995486
.cpp	0.0003275406038002984
.erd	0.0004085504034999758
./knapsack	
.c	0.00032466523080001936
.cpp	0.0003425663476002228
.erd	0.0007375786302996857

Fig. 46: Raw Values for Full Tests

./naivepatternmatch_half	
.cpp	0.00033459826950056593
.erd	0.0003902664184999594
.c	0.00028477094119953107
./bubblesort_half	
.erd	0.0003750829272001283
.cpp	0.00031711082230031026
.c	0.00027995877459979963
./primefactors_half	
.c	0.00027967877829942155
.cpp	0.0003154710809001699
.erd	0.00037827566400010253
./fibonacci_half	
.erd	0.00038300193329996547
.cpp	0.0003229927765998582
.c	0.00028791659370035633
./knapsack_half	
.c	0.0002972017915999459
.cpp	0.00032958986180019564
.erd	0.0004569238885999075

Fig. 47: Raw Values for Half Tests