DESIGN AND IMPLEMENTATION OF THE MOREHEAD-AZALEA COMPILER (MAC)

Dalton Hensley and Heba Elgazzar

Department of Engineering Sciences, Morehead State University, Morehead, KY, USA

ABSTRACT

Within the realm of computer science exists the ever ubiquitous programming language compiler. The role of the programming language compiler is that of a translation device which, among other things, must correctly and efficiently translate its source language to any number of target languages. For example, the C compiler is designed to convert its source code into executable binaries that are backed by a myriad of so-called instruction set architectures, such as x86-64. In any case, this translation process is said to be opaque to the compiler's end-user, allowing for the automation of an end-to-end source program to target language pipeline. Our goal, then, is designing and developing such a pipeline as to allow for the existence of the novel "Azalea" programming language.

KEYWORDS

Azalea, Compiler, Transpiler, Pipeline, Transformation, Static Analysis

1. INTRODUCTION

The significance of compilers within computer science can- not be understated, as their essential utility lies in the automation of translating messy human desires (via programming languages) into plain instructions (via binary formats). Offloading this error-prone and tedious work to compilers has served hu- mankind well, as the compiler has been but one step in a long line of improving programmer's productivity. The first generation of programming languages existed in absentia of compilers as we know them today. In fact, this period was most closely associated with—and relied almost exclusively on—the physical devices that ran the computer programs [1]. Engineers at the time were required to have an intimate and near-perfect under- standing of their computing machines, often directly loading in hand-prepared binary programs into memory, which were later executed by the onboard central processing unit [1]. As we move closer to the present day, one notices a particular trend: the preference towards abstraction.

Abstraction can be roughly defined as the partial stripping or total removal of unnecessary details, facts, and complexities of a thing while not altogether deleting its identifying characteristics. This is not only useful in everyday life, where humans can only absorb so much information at a time but also for computers. This realization led to the development of the second generation of programming languages. The second generation was most commonly linked with the innovation of assembly languages [1]. Their ingenious creation is most often attributed to both Andrew Booth and Kathleen Britten in their seminal work, "Coding for A.R.C." First published in 1947, Booth and Brit- ten's work describes the "Automatic Relay Computer," whose purpose was to offer a more straightforward interface (A.R.C assembly) and provide the automatic translation from its general source to a more specified target [2]. Their work had cemented the beginnings of what is now known as the assembler.

However, the assembler (and assembly languages more generally) had its disadvantages. It was possible that programs written in assembly had to have their source code completely rewritten as new computer hardware advances outpaced soft- ware's comparatively slow development. Simply put, assemblers still suffered from portability problems, as their assembly languages were directly tied to the ever-changing instruction set architectures; new machines meant outdated code! An additional layer of abstraction over assemblers, then, was required to afford generality over a multitude of assembly languages.

During this era, programmers successfully materialized the third generation of programming languages, which played a vital role in advancing programming language theory. More importantly, the entity behind that spearheaded this paradigm shift in automatic code generation [1]. This entity was (and is still) known as the compiler. To be clear, compilers can come in many different flavors and adaptations; however, for purposes relevant to this work, we shall consider the static ahead-of-time (AOT) variety as opposed to just-in-time (JIT).

For a point of comparison, the C programming language is most commonly backed by an AOT compiler. This has numerous advantages and disadvantages, but like any AOT compiler, its central feature is static compilation. One shall defer the technical details about static compilation for further discussion later in this paper. However, it is reasonable to briefly mention that the term ”static” contrasts with ”dynamic.”

Now, one arrives at the focus of this paper: the Morehead-Azalea Compiler (MAC). The goal of MAC is not all that different when compared to the likes of other contemporary com- pilers such as Haskell's Glasgow-Haskell Compiler (GHC) or Rust's *rustc* [3]. That is to say that the sole mission of MAC is the complete static analysis of Azalea programs—via a type system and other auxiliary systems—such that users can be confidently sure that their programs will not crumble under their feet vis-a`-vis segmentation faults. Another critical motivation for MAC is the automatic transformation of Azalea code into a safe subset of C. C notoriously has a ”handsoff” philosophy when it comes to providing rail guards and safety features (e.g. bounds checking), so having a means by which one can write safe Azalea code with the performance characteristics of C is highly desirable.

1.1. Goals for MAC

- Design and implement MAC as an ahead-of-time compiler. This means that MAC will never need to interface directly with the runtime of the end-user's Azalea program.
- Similarly, ensure that user faults and bugs are caught early rather than defer to the program's runtime.
- Construct and utilize an invaluable error reporting system within the compiler to allow the user clear visibility over mistakes in their code.
- Once all viably possible errors are rooted out of an Aza- lea program, proceed to transpile the source code to its equivalent in the C programming language. At this point, a backend of a C compiler will further bring down the code to the opaque level of an executable binary.
- Offer a simple-to-use integrated development environment (IDE) to allow for rapid iteration over user projects.
- Serialize the Azalea abstract syntax tree (AST) out to disk to be graphed and displayed on the user's screen.

2. RELATED WORK

Due to the enormous utility of compiler engineering, there would be an equally large corpus of scientific literature to draw upon. By all accounts, this assessment is accurate. The ori- gins of compiler engineering can be traced back to Grace Hop- per, who coined the term, although Hopper's use of the word in 1951 and its colloquial use have somewhat diverged [4]. Hop- per, in her work over the A-0 System, was rather much closer to what one might call a "linker" or "loader," as the system did not make use of the hallmark components of transformation and analysis, which are utilized in modern compilers. It wouldn't be until 1952, at the University of Manchester, when Alick Glennie would go on to co-op the word "compiler" to refer to his *Autocode* program, which compiled programs for the Manch- ester Mark 1 [4].

More germane to this body of work, however, comes in the latter half of generation three programming languages, as MAC has more in common with C than it does with either *Autocode* or A-0 System. This is to say that Azalea mostly mimics the internal code transformation and analysis pipeline of C, though with some slight differences. While MAC most certainly draws from its predecessors, one would be remiss to neglect making comparisons with its contemporaries. In order of relevance, these languages are Rust, Haskell, JAI, and Typescript.

2.1. Rust's Type Systems

Rust, much like the languages that came before it, under- stood the value in embedding types within the user's programs. While it is certainly the case that almost every third generation language implements their type system a little differently, one can say that its use within programming languages has led to an explosion in the enhancement in local and global reasoning of code. This is to say that despite the added complexity that comes with learning a type system's rules, one gets back the advantage of having erroneous programs excluded from the universal set of possible programs. This is in contrast with raw assembly which, notably, does not use nor does it differentiate between data types. Integers, floating points, and characters are represented via bytes and manipulate through instructions. This meant that one could make an unintentionally perilous mistake when using, say, indirect addressing using arguments that are not addresses! A carefully planned type system, much like Rust's, allows for the clear delineation between value and reference types [5]. Rust's types are backed by the type system, which is further backed by type checker and inference subsystems. Azalea is relevant to Rust in that it shares a similar design philosophy of separating out type checking and inference into two separate modules, allowing for a more cohesive implementation of its type system. Additionally, both Rust and Azalea share the notion of so-called "algebraic data types," allowing for greater expressibility through data structures. Figure 1 shows the type System View under Rust and Azalea.

2.2. Haskell's Error Reporting System

Both Haskell and Azalea share the belief that concrete and actionable error messages are vital to the usability of any programming language. To this end, significant work has been done to borrow the user-interface design philosophy from Haskell. One of the core sources of user errors, at least to some estimation, is an incoherence problem when using types. This is to say that both Haskell and Azalea are extremely strict on the placement and use of types. In Haskell, if a user declares that a function takes two integers, then its supplied arguments should also be of type integer. Simon Peyton-Jones, who is known for his contributions to the Haskell compiler (GHC), helped in the pivotal evolution of the quality of GHC's error messages. Specifically, in "Diagnosing Haskell Error Messages," Peyton- Jones et al. worked to improve the sometimes esoteric diagnostics generated by using a complex type system [6]. Because of their work,

Haskell enjoys not only the safety that comes with types but also the clarity that comes with expressive error reports. Azalea can be compared to Haskell in this regard, as MAC tries to catch any erroneous user input at every stage of its compilation pipeline.

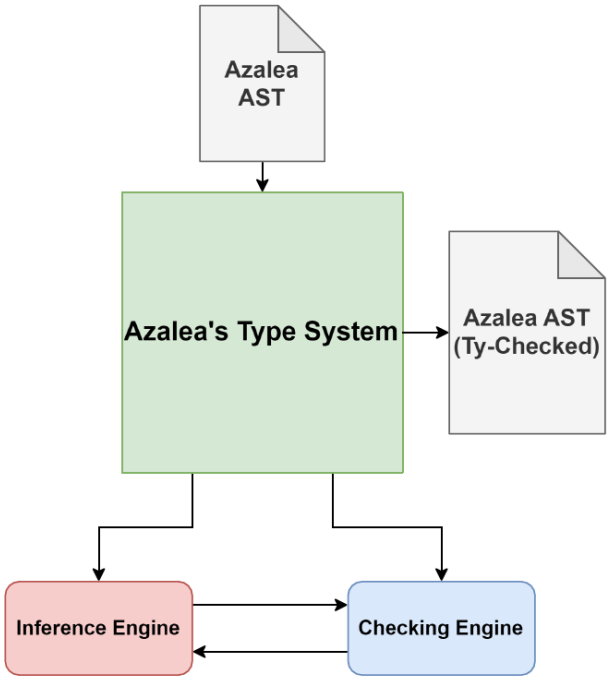

Figure 1. Type System View under Rust and Azalea

2.3. JAI's Syntax

One critical goal of Azalea is offering a user-friendly syntax that minimizes errors through consistency and expressibility.

By consistency, at least for the purposes of Azalea, one means having a syntax that repeatedly uses the same (or similar) syntactical structures through a sizable proportion of the grammar that defines it. Type qualification in Azalea—at least where required—asserts predictable syntactical expectations of having the type come after the qualifier, rather than before. This particular bit of syntax is what the internals of MAC refer to as ”declaration-based.” Having said this, it is wise to mention that Azalea's syntax borrows heavily from JAI, which is a language written by the prominent video game developer Jonathan Blow [7]. Global constants, functions, structures, and enumerations are all defined using a unified syntax, with the only considerable difference between them being the use of the keywords that tell MAC how to parse them appropriately.

By expressibility, one refers to the sizable reduction in the mental overhead one must take on in writing correct programs. Assembly runs counter to expressibility, as it may take many lines of code to write a program that prints ”hello world” to the screen. Naturally, then, expressibility can be thought of as a spectrum that correlates with a program's level of abstraction. JAI is a much higher-level language than its contemporary (C++), so it can express the same user intent with a fraction of the required lines of code.

2.4. Typescript's Transpiler

When writing a compiler, one must have a concrete plan for implementing the code generation module. Code generation gives the compiler its prime functionality, as users are typically only concerned with the end product when running their com- pilers. Contemporary programming languages have gone about this design decision in a few different ways. Languages like C and C++, for the most part, opt to target some specified instruction set architecture via a direct-tomachine-code implementation. In contrast, others find it convenient to convert to some intermediate representation (or intermediate language) such as bytecode. Rust is an excellent example, as its source is translated into quite a number of intermediate representations, which include ”high-level IR,” ”mid-level IR,” and ”LLVM IR.” This is all to emphasize the fact that transformation is a pivotal part of the overall compilation process.

The Morehead-Azalea compiler's code generation module was heavily inspired by the Typescript transpiler, formally known as *tsc*. An overview of the tsc Transpiler is shown in Figure 2. Typescript, from a programming language design perspective, is rather interesting. Its utility is predicated on the existence of JavaScript. One of the core frustrations of JavaScript is its type safety, alternatively, instead, its lack of a statically checked type system. Many runtime errors in JavaScript are, unfortunately, possible. Microsoft, who created TypeScript, recognized this immense shortcoming and saw it fit to design a JavaScript superset, including type inference and checking [8]. Designing a language around the implementation, especially in TypeScript's case, is highly valuable. Not only does it mean that one can include their features on top of an already existing language, but it also means that one gets to inherit most (if not all) of the underlying function of your target language. There is also a case to be made that transpilers, such as with the case of TypeScript, can allow developers to be much more productive, as they can be empowered by the enhancements made in the superset language.

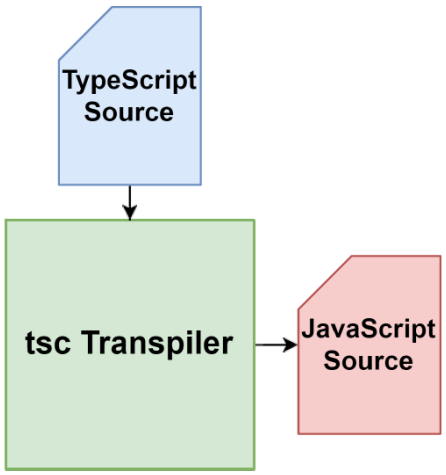

Figure 2. Overview of the tsc Transpiler

3. SOFTWARE REQUIREMENT SPECIFICATIONS

The Morehead-Azalea compiler has a few core components. However, this section splits up this topic along four different axes: Azalea's implementation language and toolchain, sup- ported platforms, language libraries, and the compiler's internal modules.

3.1. Azalea's Implementation Language

It was decided relatively early in the design process that the Rust programming language would be used to implement the Morehead-Azalea compiler. This decision was, admittedly, a nonobvious choice (as C or C++ is typically used for systems programming-related projects). Rust was chosen because it supports several modern programming paradigms that would contribute towards the rapid development of Azalea. These paradigms include derive macros, pattern matching, and a robust type system. The Rust compiler, then, is a hard requirement for anyone wishing to run their Azalea code, as the Morehead-Azalea compiler is ”prop-up” by the Rust compiler (*rustc*). It is also worth mentioning the broader Rust ecosystem via the end-to-end build tool known as *Cargo*. Cargo is also vital to the Morehead-Azalea compiler, as it allows us to build our Azalea compiler from Rust source code! Eventually, the Azalea project might become mature enough to allow MAC to be a self-hosting compiler, but that is left as a future goal.

Another essential but separate fact is that Rust is developed under the open-source model and is likewise readily available to the public. More importantly, though, is the notion that Rust is a highly portable language, which allows us to serve most platforms and architectures.

3.2.Supported Platforms

- The Windows 10 operating system is supported, though a few caveats are required to get the Morehead-Azalea com- piler running under this platform. Like C or C++, Rust re- quires the C standard library to operate correctly. Hence, Windows 10 users must have MSVC and its associated build tools.
- Similarly, MacOS is supported. Users are expected to have the ”Xcode Command Line Tools” package in order to invoke Rust via the command line.
- Finally, a Linux binary of the Morehead-Azalea compiler is also available and has a similar list of dependencies. However, all that is needed on Linux is the build-essentials package and the Rust compiler.

3.3. Language Libraries

There are a few notable language libraries that have rigid requirements for this project. Firstly, consider the libraries that Rust uses. The build process of MAC assumes that users have access to the Rust standard library and the C standard library. Both libraries are used to simplify the development process of MAC further since they heavily reduce the burden of not having to ”rewrite the wheel.” However, more specific to the topic of writing the compiler is the Ariadne library. Ariadne is a library built for the sole purpose of creating modern and production- quality error messages. These error messages are routinely used and served to the user on making a fatal error somewhere along the MAC pipeline. For example, a user may make a type error in their Azalea code; naturally, it is desired that MAC displays an appropriately formatted error report to the user's screen to inform them of the error.

The MAC project also uses a tiny amount of Python code to create an ”integrated development environment” (IDE). This form of application provides users with an easy-to-use and readily understandable interface between the non-trivial command line interface and the programming itself. The benefits of having an IDE for one's language are relatively straightforward to enumerate. Since users typically only care about writing and running their programs, having to write complex and verbose commands is usually viewed as a negative to the overall user experience. Hence, Azalea's IDE can reduce this mental bur- den by offering what is effectively a notepad with buttons.

The specific Python library that enabled the swift development of the Azalea IDE was the PyQt5 framework. This library, much like the IDE itself, is both simple to install and use since it is essentially a high-level wrapper around the C++ version. Regardless, Azalea users are required to install PyQt5 and Python (version 3.10) if they wish to take advantage of the Azalea IDE.

3.4. Azalea Compiler Pipeline

The Azalea pipeline forms the basis for the project as a whole. Without it, the compiler would be nonfunctional. This is because the pipeline takes an Azalea source file as input and then propagates this file along the various stages of compilation. One can think of the pipeline as an organized assembly line whereby transformations and analysis happen in a specific and wellregulated order. The fundamental idea, then, begins when a user finishes writing their first Azalea program. After this pivotal moment, the user will then attempt to transpile their program directly to C using our compiler. In the interim, many steps need to happen between program writing and execution. These required steps include preprocessing, scanning, parsing, semantic analysis, code generation, and code execution. Again, the pipeline process asserts that a stage can only be initiated when its dependency stages have concluded, thereby reducing any trivial opportunities for parallelism and concurrency within MAC's implementation.

4. PROPOSED DESIGNS, METHODS AND ALGORITHMS

As previously discussed, a compiler has many different forms to assume. Some compilers, such as Rust's and C's, use what is known as "ahead-of-time" compilation. This variety of compilation asserts that the majority (if not all) of the program's semantics checks and mechanical transformations will happen be- fore the program's runtime. Moreover, the end result of ahead-of-time compilation is an executable binary from which users can run their programs. Another popular choice is "just-in- time" compilation, which compiles (and recompiles) the pro- gram at its runtime. Since Azalea specifically targets C—which is an ahead-of-time language—it makes sense to favor static analysis rather than dynamic. In order to accomplish our ahead- of-time design, Azalea needed to follow a pipeline design that frontloaded all of the required transformations and analysis passes over an Azalea source file. The following subsections will delve into the design of each stage in the MAC pipeline.

4.1. Preprocessor Design

When users write their Azalea programs, they may be tempted to insert help comments that aid in understanding their code. Notably, comments do not affect their code in any way, as the composition of a comment is just supplemental text. It is prudent that these comments get stripped from the source file before they get passed to the later stages of compilation, as the comments are useless and would only obfuscate the scanner and parser. How the preprocessor strips away comments is also critical. Our current implementation consists of a nested for- loop that linearly scans for "start" and "stop" comment markers. Azalea uses the same methodology as C and C++ regarding single and multi-line comments, using "//" for the former and "/**/" for the latter. It is essential to mention that observing a single "/" is inconclusive on its own, as it may very well be a division operator token. Hence, the nested for-loop must peek at the adjacent character in order to differentiate between comments and division.

Finally, preprocessing is also vital in the detection of so- called erroneous characters. These are characters whose use is unsupported. For instance, the "@" character has zero utility in Azalea. If any instance of this character is detected, Aza- lea will generate an error diagnostic and display it to the user via the command line. Now, how this check is implemented is rather curious, as it

assumes that a set of characters exists that is valid. The easiest way to achieve this functionality is by creating a ”whitelist.” This whitelist will contain only those characters whose use is authorized. As we perform our comment stripping, we simultaneously check if the given character is in the whitelist. If it is, we proceed to the next character; otherwise, an error is thrown.

4.2. Scanner Design

Scanning is the next stage in the compilation pipeline. This stage is chiefly responsible for accumulating tokens, which are analogous to words and punctuation in everyday written languages. An overview of the Scanning Process is shown in Figure 3. The way it works is rather simple, though the very first stage in Azalea’s implementation is the enumeration of the kinds of tokens that may be observed. The types of tokens that Azalea supports are numbers (floats and integers), booleans (true or false), strings (text), keywords, operators, and identifiers. Once the varieties have been concretely established, the scanner can proceed to iterate over the character stream supplied via the preprocessing module. Our scanner goes character-bycharacter, only stopping when it hits and ambiguous character or white space. An ambiguous character is one whose interpretation depends on the next character. For ex- ample, the string ”let x = 2.3;” Has the ambiguous substring ”2.3” because the number ”2.3” is a float. Since floats depend on therebeing a number after the decimal, there may be an error if this invariant is not upheld. Floats must assert that the next character following a decimal be a number. Otherwise, an error is thrown to the user! The other interesting case is when a white space character is encountered, which means we have reached the end of the current token and can begin processing the next one. The scanning process concludes once the procedure has reached the end of character stream.

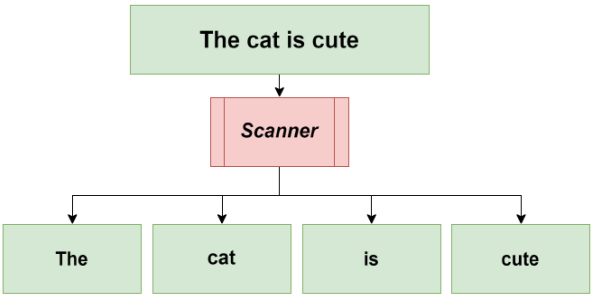

Figure 3. Overview of the Scanning Process

4.3. Parser Design

Azalea’s parser directly follows scanning, and its central job is consuming the token stream to produce an abstract syntax tree (AST). The AST of Azalea is a recursive data structure whose fields contain pointers to other nodes in the tree. Each node represents a ”production rule” from the formal grammar that specifies Azalea’s syntax. Fundamentally, there are two high level concepts with parsing Azalea code: slots and keys. Slots (or holes) are branches within the tree whose composition comprises nodes. So, the ”slot” for the variable binding will comprise a branch with five nodes. The first node will expect the ”let” keyword, as it is declares the start of the variable binding. Next, we assume that the next token will be an identifier (the binding name) followed up only with the assignment operator, expression, and semicolon. In this analogy, the ”keys” that fill the slots are the tokens from the token stream!

The exact manner in which we construct the AST is surprisingly simple, as our approach uses ”Recursive Descent” parsing. The main idea of this algorithm is to model your visitor routines around the AST itself. This means that there will be a ”visit” function for every production rule that makes up the tree. Notably, this scheme uses recursion, meaning the functions will often call themselves in order to convert the token stream into the tree.

4.4. Semantic Analyzer Design

It is highly important that our previously generated AST be free from any trivially detectable errors, as these mistakes will propagate during code generation and produce malformed C code. The semantic analyzer takes inspiration from the parser in the sense that it is entirely composed of recursive functions whose sole objective is the verifying that certain invariants are upheld. Some semantic checks are relative simple to implement, while others are significantly more involved. One that is trivial to construct is the so called ”function arity” checker, which walks the branch of a function call within the AST to verify if the number of supplied arguments matches the expect number of formal parameters of the function’s definition. If these two numbers are unequal, we must report this to the programmer as an error.

One decidedly complicated check is the type checking system, which is made up of formal ”type rules” that govern how types can be used. An example of a type checking error is shown in Figure 4. Adding two numbers together, such as adding two integers, is one such rule specified in the type system. Mixing types along the boundaries of arithmetic and relational operators is strictly forbidden, as Azalea’s type system does not incorporate implicit type conversion. If you wish to treat an integer as a float, then it is required that you use the ”as” keyword which will perform the explicit conversion for you.

One important thing to emphasize is the motivation behind the Azalea semantic analyzer. More succinctly put: why does the Azalea compiler need a semantic analyzer in the first place? Recall that the central goal is to translate Azalea source code into its C equivalent. One also knows that C, like any programming language, demands that certain rules are followed as to al- low for the successful compilation a provided program. There- fore, Azalea is motivated by the desire to catch bugs that would be rejected by the C compiler (while also checking for bugs that C does allow). In eliminating these bugs ahead-of-time, we reduce the amount of time and effort needed that would otherwise be sent debugging transplied C code. It is better (and easier) to debug your Azalea code than it is to look over the generated C code after the fact.

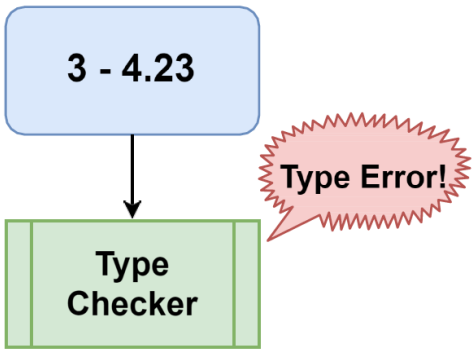

Figure 4. Example of a Type Check Error

4.5. Code Generator Design

Finally, the last core module within MAC is the code generator, which is responsible for the one-to-one translation of Azalea source code into C. Code generation is also the trick- iest module to implement, as it relies on all previous stages working correctly. Therefore, any opaque bug in the scanner will have massive ramifications, as future stages may obfuscate the issue by working with an invalid token stream! Assuming that the compiler correctly preprocesses, scans, parses, and semantically analyzes the program, one must somehow walk the validated AST to produce C code. Figure 5 shows an overview of the Azalea-to-C transpiler.

This procedure is accomplished using algorithms similar to the ones used during semantic analysis. The routines themselves may be freely copied from that module, as they allow us to visit each branch of the AST. One slight modification to these routines, though, is required. When deciding to visit a branch on the tree, it is important to perform on-the-fly writing of C strings that mimic the Azalea branch. Once a given branch has been visited, and C string produced, we then proceed to write back the string to disk via a C source file. This process is repeated until either an error is located or until we have visited every branch inside the AST.

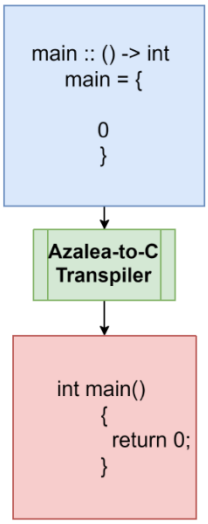

Figure 5. Overview of the Azalea-to-C Transpiler

4.6. Code Executor Design

While not technically a core module of the Azalea pipeline, the code executor serves as an optional utility that furthers the Azalea-to-C transformation by allowing for the automatic execution of the generated transpiled C file. Users interface with this feature via the command line, as it is kept separate from the default ”build” functionality.

The code executor performs its duty by first checking to see if the generated file has not moved since its creation, as Azalea must know the path to the file to execute. Once the file’s location has been resolved, it is passed into a buffer that includes the arguments and flags used when invoking the C compiler on the user’s system. A new helper thread (designated as the C compiler thread) is then spawned, which is responsible for the clean invocation and termination of the C compiler. The thread will pass the required command line arguments and flags, producing the hopefully bug-free binary executable.

5. TESTING AND DISCUSSION OF RESULTS

The testing of the Morehead-Azalea compiler is predicated on the amount of modules that make it up. Since the com- piler comprises five core components, the test coverage should mainly focus on servicing these modules. Additionally, it makes the most sense to write unit tests that individually stress modules. System testing in this case would be rather redundant, as testing code generation is effectively testing the entire system (all modules must function correctly for code generation to operate properly).

5.1. Preprocessor Testing

In order to test the compiler's preprocessor, it is important to distinction between the types errors that are expected at this stage of compilation. During preprocessing, there can only ever be two kinds of errors: incomplete code comments or erroneous character codes. Figure 6 shows an example of an error report in the event of an erroneous character code.

Figure 6. Preprocessor Unit Test

5.2. Scanner Testing

Much like in our preprocessor test, we can perform a similar evaluation my targeting bugs that are specific to the scanner. One possible (and rather subtle) bug can arise when writing floating point numbers, as they must be formatted to include a leading number, a decimal, and ending with any number of dig- its. Figure 7 shows an example demonstrating a failure during scanning.

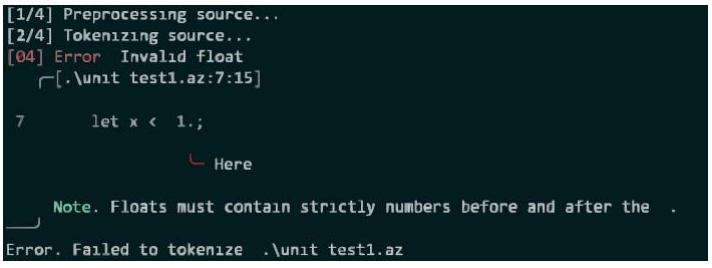

Figure 7. Scanning Unit Test

5.3. Parsing Testing

Recall that Azalea's parsing module's purpose is constructing the AST via the token stream. This process is accomplished successfully when the program's syntax is correct, and a pro- gram's syntax is correct when every token can be placed into a "slot" specified by Azalea's grammar. One possible error al- ready discussed is the possibility for misplacing a token (or omitting one

that is expected). Forgetting a semicolon at the end of a variable binding declaration is an example of a syntax error which our unit tests should cover. Figure 8 shows an example of parsing unit testing.

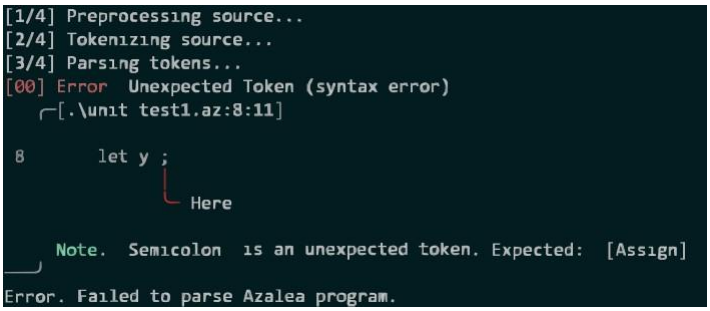

Figure 8. Parsing Unit Test

5.4. Semantic Analysis Testing

Once the Azalea AST has been constructed, the process of semantic validation beings. The procedures involved in vali- dating the AST are responsible for a few things, namely the validation of several vital invariants. One such invariant is arity checking, though this program property has already been thoroughly defined in the sections above. In order to properly test Azalea's semantic analyzer, it suffices to target areas where we expect issues to arise. Our unit tests specifically highlight the care in which users must take in order to satisfy the program's formal semantics. Based on preliminary testing, the Morehead- Azalea compiler seems to be rather robust when it comes to detecting and reporting semantic errors to the user. An example of semantic analysis unit testing is shown in Figure 9.

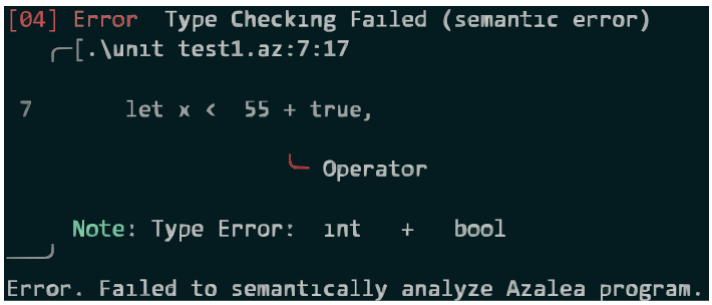

Figure 9. Semantic Analysis Unit Test

5.5. Code Generation Testing

The final module which needs to be thoroughly tested is the code generation component of the compilation pipeline. Any bugs within the code generator will directly affect users, as their Azalea code will likely be mangled and malformed. One test that exhaustively stresses every part of the code generator is to generate every code construct provided by Azalea. One can be reasonably sure that if any errors arise, it would be during the generation of an entire program (rather than any one code construct on its own). Figure 10 shows the effect of running a code generation pass over a valid Azalea AST.

Figure 10. Example of Code Generation Testing

5.6. Discussion of Results

It would seem that, at least occurring to our testing, Aza- lea can robustly and reliably determine not only when a user makes an error, but also its *kind*. This is highly important for debugging purposes where tracking a bug would be elusive. All of this is to say that each module within the Azalea pipeline is cognizant of and responsible for their errors. This allows for the ”separation of concerns” when handing errors (whether they user mistakes or internal compiler errors). Another essential thing to mention is that the user is also informed about whether their Azalea program had compiled successfully. This alone should help to eliminate any ambiguity as to if there are any unresolved problems with the Azalea compiler, one will never reach the compiler’s ”completion” check if there are ever any problems.

6. CONCLUSION AND FUTURE WORK

In order to properly conclude this project, it suffices that one first reviews the initially stated goals and determines whether or not they have been accomplished.

1. Design and implement MAC as an ahead-of-time compiler. This means that MAC will never need to interface directly with the runtime of the end-user’s Azalea program.
2. Similarly, ensure that user faults and bugs are caught early rather than defer to the program’s runtime.
3. Construct and utilize an invaluable error reporting system within the compiler to allow the user clear visibility over mistakes in their code.
4. Once all viably possible errors are rooted out of an Aza- lea program, proceed to transpile the source code to its equivalent in the C programming language. At this point, a backend of a C compiler will further bring down the code to the opaque level of an executable binary.
5. Offer a simple-to-use integrated development environment (IDE) to allow for rapid iteration over user projects.
6. Serialize the Azalea abstract syntax tree (AST) out to disk to be graphed and displayed on the user’s screen.

Beginning with the first item, the stated goal was to construct an ahead-of-time compiler that produces an executable binary after its completion. Given that our compiler can produce executable binaries via a transpiler design, one can be safe in asserting that this goal was met. There was also the hard requirement that any written Azalea bugs be made transparent to the user while providing a differentiated error message via an error reporting system. Since our testing result indicates this is a provided and expected behavior, one can checkoff goals two and three from the list.

There was also the concern of implementing our head-of- time compiler as a transpiler, as writing our machine code generation backend (with optimizer passes) was infeasible. The results section above shows that the Morehead-Azalea compiler can accurately translate Azalea source code into the equivalent C source. In reflecting on this design decision, one may realize that implementing MAC as a transpiler facilitated two core properties: fast performance at runtime and static checking prior to runtime. Not only does one get the safely features which are provided by static

analysis, but one also gets the performance characteristics of the target language (C in this case). Furthermore, Azalea supports the usage of an integrated development environment for programmers who wish to avoid the headache of the command line altogether. Users can load, save, and run programs via the IDE, which greatly accelerates the development of Azalea programs.

Finally, then, is the lesser-known feature called ”AST serialization.” This ability of the compiler refers to translating the AST into a data format that can be saved to disk. For our pur- poses, Azalea translates its AST into a JSON file, which then can be loaded into an optional Python utility program that can graph the tree.

While the stated objectives have all been accomplished, much needs to be completed. For starters, Azalea does not have first-class support for object originated programming. This goal could be achieved by adding classes, interfaces, and methods to the list of features enabled by MAC. MAC could also incorporate more specialized functions. These functions could be used for everyday mathematical operations, file IO, or array manipulation.

REFERENCES

[1] Vass, Péter. "Programming Language generations and Programming Paradigms", web.archive.org, Jan. 29, 2020, [Online], Available: https://web.archive.org/web/20200129065933/http:/www.unimiskolc.hu/~geofiz/Oktatok/vass/Generations_and_paradigms.pdf

[2] A. Debooth and K. Britten, “CODING FOR A.R.C.”, Accessed: January 5, 2025, [Online], Available: https://albert.ias.edu/server/api/core/bitstreams/d47626a1-c739-4445-b0d7cc3ef692d381/content

[3] “Home — The Glasgow Haskell Compiler,”, Accessed January 5, 2025, [Online], Available: https://www.haskell.org/ghc/

[4] M. V. Wilkes (1968), “Computers Then and Now”, *Journal of the Association for Computing Machinery*, Vol. 15, Issue. 5, pp 1-7. DOI: https://doi.org/10.1145/321439.321440

[5] K. Ferdowsi, “The Usability of Advanced Type Systems: Rust as a Case Study”, Accessed: January 5, 2025, [Online], Available: https://arxiv.org/pdf/2301.02308

[6] D. Zhang, A. Myers, D. Vytiniotis, and S. Peyton-Jones (2015), “Diagnosing Type Errors with Class”, *Proceedings of the 36th ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 12-21. DOI: https://doi.org/10.1145/2737924.2738009

[7] “JAI Programming Language Resources and Information”, *Inductive*, Accessed January 5, 2025, [Online], Available: https://inductive.no/jai/

[8] J. Scarsbrook, M. Utting, and R. Ko, (2023), “TypeScript’s Evolution: An Analysis of Feature Adoption Over Time”, Proceedings of the 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), pp. 109-114. DOI: https://doi.org/10.1109/MSR59073.2023.00027