Mnemosyne: A Functional Systems Programming Language

Hawk Weisman
Department of Computer Science
Allegheny College
weismanm@allegheny.edu
http://hawkweisman.me

November 21, 2015

Abstract

While programming languages researchers have produced a number of languages with various qualities that promote effective programming, such languages often may not be used for low-level systems programming. The automation of memory management through garbage collection often limits the use of many languages for systems programming tasks. This proposal outlines the development of a new language intended for systems programming, with a syntax inspired by that of the Lisp family of languages and providing methods of ensuring memory safety at compile-time rather than at runtime.

1 Introduction

Systems programming refers to the implementation of operating systems, device drivers, programming language standard libraries and runtime systems, and other types of software that provide services to other software rather than to the user [34, 41].

Definition 1 (Systems Programming):
The branch of programming concerned with the implementation of operating systems, utilities, and libraries that provides services to other software rather than to a human user [34].

To some, this area of programming may seem unglamorous. Software companies often find focusing on application software more profitable, and it is certainly easier to communicate the purpose of application software to the general public. However, high-quality implementation of systems software is vital to all computer applications, whether in industry, the sciences, or in the home. New advances in such systems and the continual process of improvement and maintenance of existing ones is necessary in order to continue to serve the ever increasing demands of computer applications.

The programming languages community has, in recent years, produced a number of new programming languages, such as Haskell [19, 23], Scala [36, 37], and others. These languages boast a number of qualities that support the fast and easy implementation of high-quality software, such as more expressive syntax and typing disciplines that uncover errors at compile-time. However, a majority of systems programming is still done in C [24], a language which first appeared in 1972. C is plagued by safety issues and can be very difficult to program in, especially for novice programmers or those unfamiliar with its challenges [4, 39, 41].

Why, then, do systems programmers, who work in an area where safety and correctness is often vital, use such an old and difficult language almost exclusively? There are a number of reasons behind the systems programmer's hesitation to try more modern languages. Despite the rapid, ongoing increase in the capabilities of computing hardware, performance and efficiency in both time and space are still deeply important in systems code, as small drops in performance can have major impacts on the application software that relies on a system program [41]. Applications programmers can often benefit from the use of slightly less efficient abstractions and structures if they compose well and are easily understood, but systems programmers have no such luxury.

Some of the characteristics of modern programming languages, such as automatic memory management, dynamic typing, and higher levels of abstraction, which make applications programming easier can have an inverse effect on systems programming. Automatic memory management, as discussed in more detail in Section 2.2, is a particularly major issue.

This paper proposes the implementation of a new programming language suitable for some systems programming tasks. This new language attempts to reconcile the requirements necessary for lower-level systems programming with the safety and expressive power of modern functional programming languages. Influence is drawn from the consistent and elegant syntax of the Lisp family of programming languages (Section 2.1)) and from alternative memory management methods such as those used in the Rust programming language (Section 2.3) [32].

2 Background

2.1 The Lisp Family of Programming Languages

Lisp is a family of *functional programming* languages characterized by the use of a form of prefix notation and a focus on singly-linked lists as a fundamental data structure [44, 45].

Definition 2 (Functional Programming):
A programming paradigm which models computation through the application of functions, or in which function application is the primary or the only control structure [20, 51].

Languages of the Lisp family are among the oldest computer programming languages still in use today, with FORTRAN being the only preceeding language that still sees any use. There are a number of reasons behind Lisp's enduring popularity. Perhaps one of the most fascinating such property is that of *homoiconicity*, the quality by which the textual structure of a program's source code is identical to the computer's internal representation of that program [45, 49].

In contrast to C and languages with C-like syntax, Lisps are *expression languages* rather than *statement languages*. Where C-like languages consist of both statements, which correspond to an executable action, and expressions, which evaluate to some value, Lisp programs consist only of expressions. Thus, Alan Perlis' sardonic observation that "[a] LISP programmer knows the value of everything, but the cost of nothing [38]."

In Lisp, all language constructs take the same consistent form of the *S-expression*, a parenthesized expression consisting of an operator and one or more operands, separated by spaces.

Definition 3 (S-expression):
A notation for describing nested list data structures, where each list is delimited by parenthesis characters and each list element is separated by spaces.

The following Backus-Naur Form describes a simple grammar for S-expressions:

$\langle s\text{-}exp \rangle \rightarrow$ '(' $\langle op \rangle$ $\langle exp \rangle$ ')'

$\langle exp \rangle \rightarrow \langle s\text{-}exp \rangle \mid \langle constant \rangle \mid \langle exp \rangle \ \langle exp \rangle \mid \varepsilon$

$\langle op \rangle \rightarrow \langle language\text{-}function \rangle \mid \langle user\text{-}function \rangle$

Listing 1: Backus-Naur form for a S-expression grammar [44].

A number of symbols (i.e., those for constants and functions) are left undefined, as this grammar is not intended as a complete description of a language.

For example, to sum two numbers, one would state:

```
(+ 1 1)
```

More complex concepts may be expressed just as easily with these S-expressions. Consider a conditional logic expression:

```
(if (> a 0)
  (+ a 1)
  (- a 1)
)
```

This expression evaluates to the value of `a` + 1 if variable `a` is greater than zero, and `a` - 1 if it is not.

The *Von Neumann architecture*, in which the same memory is used to store both data and instructions [50], is fundamental to the history of modern computing. By storing program code and data in the same consistent list structure and allowing programs to act easily on their own instructions [42], Lisp is something of a reinforcement of this fundamental design.

This quality of homoiconicity has been observed to promote methods of thinking about code and its execution that can be beneficial to high-quality programming [40, 45]. Since the source code as understood by the programmer is structurally equivalent to the program as understood by the compiler or interpreter, there is less of a disconnect between person and machine [45]. The primary goal of programming language design, in this author's opinion, is to find a comfortable middle ground between two radically different forms of mind: that of the human programmer, whose mind is very small, very slow, and very good at abstract reasoning and at handling ambiguity; and that of the computer, whose mind is very large and fast, but must be given very precise instructions. Lisp's homoicionic syntax seems a good step towards this symbiosis.

2.2 Garbage Collection

Differing methods for memory management is one of the primary reasons that systems programmers refrain from using more modern programming languages. A majority of modern languages provide automatic memory management in the form of *garbage collection*. C, on the other hand, requires the user to manage memory manually, by allocating it with `malloc()` and then deallocating it with `free()` when it is no longer needed.

Definition 4 (Garbage Collection):
A form of automatic memory management which attempts to deallocate memory objects automatically at runtime, typically by tracking references to those objects. [3, 8].

Garbage-collected languages, such as Java, Scala, and Python, make the implementation of application software much easier by freeing programmers from having to concern themselves with the responsibility of manually allocating and deallocating memory. This reduces the cognitive load on the programmer and decreases the risk of memory leaks and dangling pointer bugs [8, 15]. However, in order to deallocate the memory objects that are no longer in use, the program must first find these objects. This necessitates some form of analysis (e.g., *tracing* or *reference counting*) to determine which memory objects can be deallocated, which therefore requires, at minimum, a complete traversal of the program's memory space [3, 8, 15].

While a great deal of work has taken place in order to make garbage collection as fast and efficient as possible, the measurable costs associated with a garbage collector pass [15] are unavoidable. While memory allocation is never completely free — even requesting memory from *malloc()* requires some work to select which blocks ought to be allocated — the garbage collector requires significantly more time to perform its analysis. Furthermore, the work of manual allocation occur only when memory is allocated or deallocated, while the garbage collector typically needs to periodically interrupt the program's execution to collect garbage. This means that a programmer using manual memory allocation can, with some thought, optimize their programs by avoiding allocating new storage during some performance-critical functions or code segments.

For application programmers, the benefits of garbage collection are worth the performance tradeoffs, but for systems programming, this is often not the case — if a garbage collector pass interrupts a time-critical system component such as the OS scheduler, the performance of all programs running on the system is impacted. Furthermore, low-level systems components must often function with minimal support from a runtime infrastructure which provides features such as garbage collection, as they *are* this infrastructure for other software.

Received wisdom therefore holds forth that one cannot reconcile automatic memory management with the requirements of systems software, condemning the implementors of systems to rely on `malloc()` and `free()` forever.

2.3 Alternative Memory Management

While many would suggest that a programming language may have either manual memory management or garbage collection, there are, in fact, a number of additional options.

Definition 5 (Stack Allocation):
A method of memory management in which all memory allocation created within a scope such as a function are automatically deallocated when the flow of program execution exits that scope [6, 13].

In *stack allocation*, all memory objects are allocated on a stack, with each level in the stack (called a *stack frame or allocation record*) corresponding to a scope or execution context, such as a function. When a scope is exited, all of the memory objects allocated in that scope are automatically deallocated. This is how most languages allocate parameters to function calls. When this approach is applied to all memory allocation, it provides the same guaranteed memory safety as garbage collection, but requires significantly less work to be performed at runtime [6, 13]. However, allocating all data on the stack has the obvious flaw that there is no way to share data outside the stack frame in which it was created, placing significant constraints on the programmer and limiting the expressiveness of the language.

Rust, a new language intended for systems programming, introduces a unique solution for this issue by adding to the language a notion of *ownership*. In Rust, a memory object is said to be 'owned' by the scope in which it was allocated, placing it on that scope's stack frame. Rust then permits the owner of a memory object to either transfer ownership to another scope (called a 'move') or to share 'borrowed' references to that object. These borrowed pointers are lexically scoped, and may not be referenced after the object to which they point passes out of scope. Rust also differentiates between mutable and immutable borrows, permitting only one mutable borrow of an object at any point in time. This prevents issues related to concurrent modification. The Rust compiler includes a component called the *borrow checker*, which performs analysis of borrowing and ownership at compile-time [32]. This system provides Rust programs with guaranteed memory safety, but does not require programmers to manually allocate and deallocate objects. Essentially, the borrow checker moves the overhead of performing analysis for automatic memory management from runtime to compile-time, adding a step to the compilation process but allowing the resultant binaries to run without garbage collection.

Definition 6 (Linear Allocation):
A method of memory management in which each allocated memory object may be accessed only a single time. A linear type is produced once and consumed only once, always having a reference count of exactly 1 [1, 2, 14].

Linear allocation is a similar but more constrained approach. Linear types may be accessed only once over their lifetimes [1, 2, 14]. While this may initially seem unpleasantly limiting, we should take note of the fact that such a type would therefore require no garbage collection [1, 2]. In the fantastically-titled papers "Lively Linear Lisp: "Look Ma, No Garbage!"" (1992) and "'Use-once' Variables and Linear Objects: Storage Management, Reflection and Multi-threading" (1995), Baker described a type system for programming languages with linear allocation which differentiates shared memory objects, which may be accessed outside of their parent scopes, from unshared objects. Once an object is made public, it may be used only once [1, 2]. The similarities between Baker's linear types and Rust's ownership are obvious: both methods attempt to encode information about memory allocation in a language's type system and, in doing so, perform automatic memory management in the compiler rather than at runtime.

3 Method of Approach

This proposal outlines a strategy for the creation of a new language intended for systems programming, called Mnemosyne, after the Titaness who personified the concept of memory in ancient Greek mythology. The primary goal of the Mnemosyne project is to create a safe, expressive functional programming language which is nonetheless suitable for high-performance and systems programming. Like Rust, Mnemosyne aims to fulfill a niche similar to that of C++, offering similar performance and low-level access to hardware, but providing some abstractions to ease programming.

While Rust and Mnemosyne have similar goals, Rust is among the first languages with lifetime-based memory management to gain widespread adoption. Thus, it seems that there is still a need for increased diversity of languages in this design space. Rust and Mnemosyne differ along a number of axes. Rust's syntax is inspired primarily by that of C and the languages descended from it, while Mmemosyne's is drawn from the Lisp family. Many programmers find Lisp-style homoiconic expressions to be powerful, elegant, and expressive, as discussed in Section 2. Furthermore, future Mnemosyne development is planned to include features not present in Rust. For example, in a future release, Mnemosyne will enforce function purity, as in Haskell [17, 19, 23], which will eventually be developed into a fully featured effect system in which the side effects of functions are encoded in the type system [31].

3.1 Design Considerations

3.1.1 Syntax

Mnemosyne borrows the expression syntax of the Lisp family of languages, as described in Section 2.1. S-expression syntax has a number of advantages. In addition to assisting programmers in understanding the compiler or interpreter's understanding of their code (through the concept of homoiconicity), the consistency of the syntax of S-expression based languages makes parsing fairly simple. Ease of parsing is of great advantage to the implementation of proof-of-concept or research programming languages, as developing a parser is a significant portion of the workload of compiler implementation.

Despite its S-expression-based syntax, Mnemosyne is not technically a member of the Lisp family. In addition to the use of S-expressions, Lisps are generally characterized as being dynamically-typed interpreted languages. Since a primary goal of the new language's design is to perform memory management at compile-time, it would therefore have to be a compiled language. Furthermore, in order to permit reasoning about memory at compile-time, a change in typing discipline is also necessary, since the type of a value determines the amount of space that it occupies in memory.

Also unlike other Lisps, functions and anonymous functions in Mnemosyne are defined using *pattern matching*, similar to the Haskell programming language [17, 19, 23]. Pattern matching is a syntactic construct common to functional programming languages such as Haskell [17, 19, 23], Scala [36, 37], and ML [25, 29]. In pattern matching, an expression or value is tested against a series of pattern expressions, which can test equality for constant values, test subtype relationships,

and destructure algebraic data types. The compiler converts these match expressions to decision trees in the resultant program binary, resulting in a high performance and expressive method of program flow control [25, 29, 30, 47] In this approach to function definition, a function or lambda is defined by one or more *equations*, consisting of a pattern expression and a function body. The inputs to the function are matched against each pattern expression until a match is found. Any variables present in the pattern expression are then bound to the appropriate arguments, and the corresponding function body for that equation is then evaluated and returned [19, 23].

Function definition through pattern matching was chosen for Mnemosyne for a number of reasons. Primarily, it encourages programmers to consider functions as they are in mathematics: a mapping of input values to output values; and encourages the use of explicit base cases for recursive functions [17]. Additionally, function definition through pattern matching permits the programmer to write functions with varying arities depending on the passed arguments, a useful tool in languages which provide automatic currying[1], a feature planned for a future Mnemosyne release.

Additionally, there would be a need to introduce syntax for working with pointer types. Lisps typically do not provide a great deal of functionality for performing operations such as pointer arithmetic. In a systems programming language, methods of working on memory addresses at a low level are necessary. Furthermore, depending on the memory management method employed, there may additionally be a need to differentiate between types of references, as in Rust, which distinguishes between pointers or references which are borrowed from another scope, those which were moved from another scope, and those which are owned by the scope in which they are referenced [32].

3.1.2 Semantics

Since safe and efficient memory management is a major design goal, it is necessary to consider the memory semantics of Mnemosyne programs very thoughtfully. Thus, developing memory management methods for Mnemosyne will require additional research and development.

Lisp and Lisp-like programming languages are in some ways uniquely well-suited to scope-based methods of managing memory; Scheme [44] (a Lisp variant) was one of the first lexically-scoped programming languages, and the S-expression syntax of Lisps make scopes explicit to the programmer. A system for memory management based on compile-time analysis of scopes seems possible.

The linear approach, as described by Baker [1, 2] and Hawblitzel et al. [14], would be particularly simple to implement and embraces the functional programming philosophy of immutable data structures. However, a language based around immutability may be insufficiently low level to meet the needs of systems programmers, and the overhead of copying objects may be unacceptable in real-time programs such as an operating system kernel. A memory management model based on stack allocation seems to find a happy medium between safety, performance, and ‘closeness to the machine’. In order to make such a language expressive enough to be useable,

[1] As in Haskell.

a Rust-like system of ownership and lending seems necessary. Previous research [43] suggests that lifetime analysis for Lisp programs is certainly possible.

The name 'Lisp' was originally an abbreviation for 'LISt Processing', and the singly-linked list forms the core construct of all true Lisps. However, that data structure is oftentimes too high level for the needs of systems programmers. Therefore, robust syntax for working with arrays and algebraic data types must also be provided. Developing syntax to express the many new concepts introduced in the language in a manner that is unambiguous to the programmer and to the compiler, and that blends well with the S-expression syntax, may require some effort.

An additional semantic consideration that improves program safety considerably is explicit differentiation between nullable and non-nullable references. In C and most C-derived programming languages, such as C++, C#, and Java, there exists a special constant called `null`. Variables can be assigned to this constant in order to indicate that the value is unknown, such as when it has not yet been determined or is unavailable as the result of an error. However, when `null` is present in a program, it is necessary to check frequently whether or not a variable is `null`, as attempting to dereference a pointer to `null` will result in a run-time error. It has been observed that null reference errors of this form are among the most frequent faults in these languages [5, 9, 10], to the extent that the originator of the `null` reference, Sir C.A.R. Hoare, referred to it as his "billion-dollar mistake" [16].

An alternative to the `null` value, and the need for constant checking it implies, is the technique of encoding at the type level whether or not a value is nullable [10]. In this technique, the language provides a special container type for values which may not be present, and enforces that all other values not be nullable. This approach has the advantage that null reference errors can often be detected at compile time, allowing them to be resolved by the programmer rather than released into production software [10]. A number of popular functional programming languages provide such a type: Scala and Rust call it `Option` [32, 36, 37] while Haskell calls it `Maybe` [17, 23]. In order to avoid Tony Hoare's "billion-dollar mistake", Mnemosyne will follow their example. Special syntax may be provided for the `Optional` type, in order to make its use less challenging for programmers.

3.2 Implementation

The Mnemosyne compiler, called `mn`[2], will be implemented in the Rust programming language. Rust is a safe, functional programming language intended for systems programming; as such, it has similar goals as Mnemosyne, and provides an excellent platform for implementing the compiler until the language is complete enough to be self-hosting.

Programs will be compiled to machine code binaries using LLVM [26], a project which provides infrastructure for programming language compilers. LLVM primarily provides functionality related to the *backend*, or code-generation, component of a compiler. It provides an intermediate representation (IR) format which the frontend components of a compiler can generate, and can perform a number of common optimizations on that IR [26, 48]. Additionally, LLVM is capable of generating machine code for a very wide range of operating systems and processor architec-

[2]Pronounced "Manganese".

tures [26, 48], a major advantage that was directly responsible for the choice of LLVM IR as a primary compilation target for Mnemosyne. As the open-source Rust compiler uses LLVM for code generation, there are pre-existing Rust bindings for the LLVM API, making the implementation of the compiler backend much easier.

The Mnemosyne syntax analysis stage will consist of a parser with integrated lexical analysis (tokenization). While many compilers include separate tokenizer and parser stages, the simplicity of Mnemosyne's Lisp-inspired grammar should permit the parser to generate abstract syntax tree nodes from the program character string directly, without a separate lexical analysis phase. The parser will be implemented using the *combinator parsing* technique [11, 12, 21, 46], a common method of implementing parsers in functional programming languages.

In combinator parsing, a `parser` is a function which either accepts or rejects a character or string of characters. A *parser combinator* is a higher-order function which takes as parameters one or more parsers, and returns a new parser that combines or modifies the input parsers according to some rule. Common parser combinators might include a repetition combinator that parses one or more repetitions of a character or string, a disjunction combinator which parses either one character or string or another, and a sequential combinator that parses one character or string followed by another. By combining many small parsers using these combinators, a recursive-descent parser is implemented [7, 21, 46]. Unlike the often nightmarishly complex code generated by traditional parser generators such as `yacc` or `bison`, parsers written using combinator parsing are implemented entirely by the programmer, and are often much simpler and easier to understand. Combinator parsing libraries exist in a number of functional programming languages, such as Scala [33] and Haskell [27]; Mnemosyne will use a Rust library called `combine`, based on Haskell's `Parsec` [27].

```
1  fn parse_def(&self, input: State<I>) -> ParseResult<Form<'a, U>, I> {
2
3      let function_form // The function definition form consists of...
4          = self.name() // ... an identifier...
5                .and(self.function()) // ...followed by a function body
6                .map(|(name, fun)| // convert parse results to AST node
7                      DefForm::Function { name: name
8                                        , fun: fun });
9
10     let top_level // The top-level def form consists of...
11         = self.name() // ...an identifier...
12               .and(self.type_name()) // ...and type annotation...
13               .and(self.expr()) // ...followed by some expression.
14               .map(|((name, ty), body)| // convert results to AST node
15                 DefForm::TopLevel { name: name
16                                   , annot: ty
17                                   , value: Rc::new(body) });
18
19     // A 'def' statement consists of...
20     self.reserved("def") // ...the 'def' keyword...
21         .or(self.reserved("define")) //...or the 'define' keyword ...
22         .with(function_form.or(top_level)) // ...and a definition.
23         .map(Form::Define) // convert parse results to AST node
24         .parse_state(input)
25 }
```

Listing 2: Example of combinator parsing in the Mnemosyne compiler.

Listing 3.2 shows a code snippet from the Mnemosyne parser that demonstrates the use of combinator parsing. In the example, a parser for the `def` keyword is defined. The parser begins by defining, on lines 3 and 10, smaller parsers for the function and top-level global definition forms of the `def` expression. These parsers call other parser functions, and collect the results of those parsers into the appropriate abstract syntax tree node, using the `and` combinator to string together two smaller parsers. Then, on lines 20 and 21, the `def` form is defined as being either the keyword `def` or the keyword `define`, followed by either the top-level definition or function definition parsers. This demonstrates the use of the `or` combinator, which accepts the result of either of two combined parsers, and `with` combinator, which functions like the `and` combinator, but discarding the result of the first parser rather than passing it to the subsequent parser. Finally, the `map` function is used to apply another function or closure to a parse result. Here, on lines 6, 14, and 23, it is used to wrap the parse results of various parsers in the appropriate AST node structure.

The Mnemosyne compiler is designed with the anticipation that most Mnemosyne libraries will be distributed as source code rather than as compiled binary files. Library sources will be compiled alongside the programs that depend on them, and cached locally so that they need only be

recompiled in the event of changes. This approach frees library developers from having to maintain and publish multiple binaries for various operating systems and configurations. Additionally, it may allow the compiler to output smaller binaries in many cases, as code from dependencies that is not used in the program being compiled can often be elided from the result executable. Finally, client developers will be able to opt in or out of various library features using conditional compilation.

However, it will often be necessary to link against compiled binaries as well, such as when interacting with operating system APIs or libraries written in other languages through the foreign function interface. Since Mnemosyne is intended for systems programming, this may occur frequently. Therefore, the compiler will also have a facility for linking against binary object files.

4 Evaluation Strategy

Assessing and evaluating a new programming language with any level of rigor is a fairly daunting task. An exhaustive evaluation of any language is likely to require multiple independent research studies over a long period of time. Such a large-scale evaluation is also likely not possible unless a significant amount of software has been implemented in the language being investigated. However, since this proposal describes the creation of a new language, it is necessary to make at least a preliminary attempt to evaluate the language and determine if it fulfills its design goals.

There are a number of methods by which the quality of a language may be assessed. Ideally, empirical studies would be performed to measure the quality of software implemented in the language being studied [4, 39], the productivity of programmers working in that language [18], and the opinions or subjective experiences of those programmers. These studies may take the form of controlled experiments where programmers are asked to implement a specification in multiple languages [18, 39], studies which observe and compare existing open-source software [4, 39], and studies which conduct surveys of the opinions of programmers regarding various languages [39]. However, this form of large scale empirical research is considered unfeasible due to time considerations and the lack of widespread adoption of Mnemosyne at this point in its development. While it would be ideal to conduct large-scale user studies such as the Naval Surface Warfare Center study discussed by Hudak and Jones or the analysis of open-source software conducted by Ray et al., the initial Mnemosyne prototype proposed here will not have reached an appropriate level of development for this form of evaluation.

The efficacy of a majority of the language design choices, such as lifetime analysis, and implementation strategies, such as combinator parsing, employed by Mnemosyne have been demonstrated in previous literature. Therefore, our primary goal in assessing Mnemosyne, then, is not to determine whether these techniques are effective, but to ensure that they have been implemented correctly. To wit, our objective is to conduct *program verification* rather than *program validation*.

Definition 7 (Program verification):
The act or process of testing a software program to ensure that it conforms to a specification, requirement, or regulation. In contrast with program validation, verification is concerned with ensuring that the program implements the specification or requirement correctly, rather than whether or not the specification or requirement is correct [22, 28].

Verification of Mnemosyne will consist of a number of phases: unit testing, integration testing, performance testing, and demonstration. Unit testing refers to the practice of writing short pass/fail tests that assess whether or not a single function or code unit behaves as expected. This is used to verify individual system components. Integration testing refers to the similar practice of writing larger tests that assert that multiple components of the system interact to produce expected behavior. Performance testing measures the execution time of functions, code units, and system tasks in order to assess the system's performance. All of these testing techniques will be used extensively during Mnemosyne's development. Unit tests will frequently be written prior to the implementation of the system components under test, in order to serve as a specification written in code rather than in words.

Finally, the demonstration phase will consist of writing sample programs in Mnemosyne that demonstrate its functionality, and assessing whether these programs are correctly compiled. While the prototype Mnemosyne implementation may not be complete enough for implementing whole software systems, segments of these systems could be written and assessed. Potentially, the beginnings of the Mnemosyne standard library could be used for testing the compiler's correctness and performance.

5 Conclusion

The programming language research community has, over the years, produced a great many advances in language technology. Many of the languages commonly used in the early days of programming as a discipline have been left behind in the mists of time — despite their great influence on the programming languages in use today, nobody really uses ALGOL or Pascal these days. Two language families, however, have refused to die: the Lisp family and the C family.

Why have these two obstinate branches of the family tree of programming languages hung on to life so stubbornly? This author submits that the abiding popularity of Lisps is because there is something fundamentally valuable in the cognitive model of computation they provide, and respectfully reminds the reader that he is certainly not the first to do so [40, 45].

C, on the other hand, seems to have stuck around not because it is good, but because it is a necessary evil [41]. Systems programmers do not use C and its offspring because they find those languages pleasant and enjoyable to use; searching publically available source code archives for vile language of the reader's choice and analyzing the languages in which such words and phrases occur most frequently provides strong evidence of this fact[3]. The difficulty of programming in C, and the prevalence of errors in C programs have frequently been observed [4, 39, 41], both by academic researchers and in the oral traditions of programmers. However, many of the features and qualities of more modern languages that make them easier and safer to program in limit their effectiveness for the sort of low-level programming which systems programmers are called upon to do. One primary example of this is garbage collection, which greatly reduces errors in memory management and frees programmers from having to worry about memory allocation, but also makes a language unsuitable for a large amount of systems programming tasks.

[3] I invite the reader to partake of such an activity at their own discretion.

In this proposal, we have evaluated a number of methods of ensuring memory safety without the use of garbage collection, and discussed how they may be applied to a Lisp-like programming language. Furthermore, we have considered other design concerns necessary for systems programming in such a language. Finally, we have discussed how such a language might be implemented, and methods for evaluating its effectiveness. While this new language may not replace C once and for all, continued research into new languages for systems programming is vital to the health of computing as a discipline, since systems software has a great impact on all applications of computing technology.

In closing, we should consider that the idea that providing programmers with a diversity of languages and tools is vital for the continued health of the discipline. It has often been observed that learning a diverse range of programming languages contributes greatly to the individual programmer's intellectual growth [35, 40]. Furthermore, a diverse range of projects with similar goals is often advantageous to all: both competition between projects and collaboration or idea-sharing often accelerates the development of new ideas and encourages the pursuit of excellence.

References

[1] Henry G. Baker. "Lively Linear Lisp: "Look Ma, No Garbage!"" In: *SIGPLAN Not.* 27.8 (Aug. 1992), pp. 89–98. ISSN: 0362-1340. DOI: 10.1145/142137.142162. URL: http://doi.acm.org/10.1145/142137.142162 (cit. on pp. 5, 7).

[2] Henry G. Baker. "'Use-once' Variables and Linear Objects: Storage Management, Reflection and Multi-threading". In: *SIGPLAN Not.* 30.1 (Jan. 1995), pp. 45–52. ISSN: 0362-1340. DOI: 10.1145/199818.199860. URL: http://doi.acm.org/10.1145/199818.199860 (cit. on pp. 5, 7).

[3] David H. Bartley. "Garbage Collection". In: *Encyclopedia of Computer Science*. Chichester, UK: John Wiley and Sons Ltd., pp. 743–744. ISBN: 0-470-86412-5. URL: http://dl.acm.org/citation.cfm?id=1074100.1074419 (cit. on p. 4).

[4] Pamela Bhattacharya and Iulian Neamtiu. "Assessing Programming Language Impact on Development and Maintenance: A Study on C and C++". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 171–180. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985817. URL: http://doi.acm.org/10.1145/1985793.1985817 (cit. on pp. 1, 11, 12).

[5] Patrice Chalin and Perry R. James. "Non-null References by Default in Java: Alleviating the Nullity Annotation Burden". In: *Proceedings of the 21st European Conference on Object-Oriented Programming*. ECOOP'07. Berlin, Germany: Springer-Verlag, 2007, pp. 227–247. ISBN: 3-540-73588-7, 978-3-540-73588-5. URL: http://dl.acm.org/citation.cfm?id=2394758.2394776 (cit. on p. 8).

[6] Erik Corry. "Optimistic Stack Allocation for Java-like Languages". In: *Proceedings of the 5th International Symposium on Memory Management*. ISMM '06. Ottawa, Ontario, Canada: ACM, 2006, pp. 162–173. ISBN: 1-59593-221-6. DOI: 10.1145/1133956.1133978. URL: http://doi.acm.org/10.1145/1133956.1133978 (cit. on p. 5).

[7] Nils Anders Danielsson. "Total Parser Combinators". In: *SIGPLAN Not.* 45.9 (Sept. 2010), pp. 285–296. ISSN: 0362-1340. DOI: 10.1145/1932681.1863585. URL: http://doi.acm.org/10.1145/1932681.1863585 (cit. on p. 9).

[8] Edsger W. Dijkstra et al. "On-the-fly Garbage Collection: An Exercise in Cooperation". In: *Commun. ACM* 21.11 (Nov. 1978), pp. 966–975. ISSN: 0001-0782. DOI: 10.1145/359642.359655. URL: http://doi.acm.org/10.1145/359642.359655 (cit. on p. 4).

[9] Bob Duff. "Gem #23: Null Considered Harmful". In: *Ada Lett.* 29.1 (Mar. 2009), pp. 25–26. ISSN: 1094-3641. DOI: 10.1145/1541788.1541792. URL: http://doi.acm.org/10.1145/1541788.1541792 (cit. on p. 8).

[10] Manuel Fähndrich and K. Rustan M. Leino. "Declaring and Checking Non-null Types in an Object-oriented Language". In: *SIGPLAN Not.* 38.11 (Oct. 2003), pp. 302–312. ISSN: 0362-1340. DOI: 10.1145/949343.949332. URL: http://doi.acm.org/10.1145/949343.949332 (cit. on p. 8).

[11] Jeroen Fokker. "Functional parsers". In: *Advanced Functional Programming*. Springer, 1995, pp. 1–23 (cit. on p. 9).

[12] Richard A Frost, Rahmatullah Hafiz, and Paul Callaghan. “Parser combinators for ambiguous left-recursive grammars”. In: *Practical Aspects of Declarative Languages*. Springer, 2008, pp. 167–181 (cit. on p. 9).

[13] Chris Hanson. “Efficient Stack Allocation for Tail-recursive Languages”. In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. LFP ’90. Nice, France: ACM, 1990, pp. 106–118. ISBN: 0-89791-368-X. DOI: 10.1145/91556.91603. URL: http://doi.acm.org/10.1145/91556.91603 (cit. on p. 5).

[14] Chris Hawblitzel et al. “Low-level linear memory management”. In: *Proceedings of the 2nd workshop on Semantics, Program Analysis and Computing Environments for Memory Management*. 2004 (cit. on pp. 5, 7).

[15] Matthew Hertz and Emery D. Berger. “Quantifying the Performance of Garbage Collection vs. Explicit Memory Management”. In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA ’05. San Diego, CA, USA: ACM, 2005, pp. 313–326. ISBN: 1-59593-031-0. DOI: 10.1145/1094811.1094836. URL: http://doi.acm.org/10.1145/1094811.1094836 (cit. on p. 4).

[16] Tony Hoare. “Null references: The billion dollar mistake”. In: *Presentation at QCon London* (2009) (cit. on p. 8).

[17] Paul Hudak and Joseph H Fasel. “A gentle introduction to Haskell”. In: *ACM Sigplan Notices* 27.5 (1992), pp. 1–52 (cit. on pp. 6–8).

[18] Paul Hudak and Mark P. Jones. *Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity*. Research Report YALEU/DCS/RR-1049. New Haven, CT: Department of Computer Science, Yale University, 1994 (cit. on p. 11).

[19] Paul Hudak et al. “Report on the programming language Haskell: a non-strict, purely functional language version 1.2”. In: *ACM SIGPLAN notices* 27.5 (1992), pp. 1–164 (cit. on pp. 1, 6, 7).

[20] John Hughes. “Why functional programming matters”. In: *The Computer Journal* 32.2 (1989), pp. 98–107 (cit. on p. 2).

[21] Graham Hutton and Erik Meijer. *Monadic parser combinators*. Tech. rep. NOTTCS-TR-96-4. 1996. URL: http://eprints.nottingham.ac.uk/237/ (cit. on p. 9).

[22] “IEEE Draft Guide: Adoption of the Project Management Institute (PMI) Standard: A Guide to the Project Management Body of Knowledge (PMBOK Guide)-2008 (4th edition)”. In: *IEEE P1490/D1, May 2011* (2011), pp. 1–505. DOI: 10.1109/IEEESTD.2011.5937011 (cit. on p. 11).

[23] Simon L Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003 (cit. on pp. 1, 6–8).

[24] Brian W Kernighan, Dennis M Ritchie, and Per Ejeklint. *The C programming language*. Vol. 2. Prentice-Hall Englewood Cliffs, 1988 (cit. on p. 1).

[25] Neelakantan R. Krishnaswami. “Focusing on Pattern Matching”. In: *SIGPLAN Not.* 44.1 (Jan. 2009), pp. 366–378. ISSN: 0362-1340. DOI: 10.1145/1594834.1480927. URL: http://doi.acm.org/10.1145/1594834.1480927 (cit. on pp. 6, 7).

[26] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO ’04. Palo Alto, California: IEEE Computer Society, 2004, pp. 75–. ISBN: 0-7695-2102-9. URL: http://dl.acm.org/citation.cfm?id=977395.977673 (cit. on pp. 8, 9).

[27] Daan Leijen and Erik Meijer. “Parsec: Direct style monadic parser combinators for the real world”. In: (2002) (cit. on p. 9).

[28] Ralph L. London and Daniel Craigen. “Program Verification”. In: *Encyclopedia of Computer Science*. Chichester, UK: John Wiley and Sons Ltd., pp. 1458–1461. ISBN: 0-470-86412-5. URL: http://dl.acm.org/citation.cfm?id=1074100.1074730 (cit. on p. 11).

[29] Luc Maranget. “Compiling pattern matching to good decision trees”. In: *Proceedings of the 2008 ACM SIGPLAN workshop on ML*. ACM. 2008, pp. 35–46 (cit. on pp. 6, 7).

[30] Luc Maranget. “Warnings for pattern matching”. In: *Journal of Functional Programming* 17.03 (2007), pp. 387–421 (cit. on p. 7).

[31] Daniel Marino and Todd Millstein. “A Generic Type-and-effect System”. In: *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*. TLDI ’09. Savannah, GA, USA: ACM, 2009, pp. 39–50. ISBN: 978-1-60558-420-1. DOI: 10.1145/1481861.1481868. URL: http://doi.acm.org/10.1145/1481861.1481868 (cit. on p. 6).

[32] Nicholas D. Matsakis and Felix S. Klock II. “The Rust Language”. In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*. HILT ’14. Portland, Oregon, USA: ACM, 2014, pp. 103–104. ISBN: 978-1-4503-3217-0. DOI: 10.1145/2663171.2663188. URL: http://doi.acm.org/10.1145/2663171.2663188 (cit. on pp. 2, 5, 7, 8).

[33] Adriaan Moors, Frank Piessens, and Martin Odersky. *Parser combinators in Scala*. Tech. rep. Katholieke Universiteit Leuven, 2008 (cit. on p. 9).

[34] Thomas Narten. “Systems Programming”. In: *Encyclopedia of Computer Science*. Chichester, UK: John Wiley and Sons Ltd., pp. 1739–1741. ISBN: 0-470-86412-5. URL: http://dl.acm.org/citation.cfm?id=1074100.1074850 (cit. on p. 1).

[35] Peter Norvig. “Teach yourself programming in ten years”. In: *URL http://norvig. com/21-days. html# answers* (2001) (cit. on p. 13).

[36] Martin Odersky et al. *An overview of the Scala programming language*. Tech. rep. 2004 (cit. on pp. 1, 6, 8).

[37] Martin Odersky et al. *The Scala language specification*. 2004 (cit. on pp. 1, 6, 8).

[38] Alan J. Perlis. “Special Feature: Epigrams on Programming”. In: *SIGPLAN Not.* 17.9 (Sept. 1982), pp. 7–13. ISSN: 0362-1340. DOI: 10.1145/947955.1083808. URL: http://doi.acm.org/10.1145/947955.1083808 (cit. on p. 2).

[39] Baishakhi Ray et al. “A Large Scale Study of Programming Languages and Code Quality in Github”. In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 155–165. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635922. URL: http://doi.acm.org/10.1145/2635868.2635922 (cit. on pp. 1, 11, 12).

[40] Eric S Raymond. “How to become a hacker”. In: *Database and Network Journal* 33.2 (2003), pp. 8–9 (cit. on pp. 3, 12, 13).

[41] Jonathan Shapiro. “Programming Language Challenges in Systems Codes: Why Systems Programmers Still Use C, and What to Do About It”. In: *Proceedings of the 3rd Workshop on Programming Languages and Operating Systems: Linguistic Support for Modern Operating Systems*. PLOS ’06. San Jose, California: ACM, 2006. ISBN: 1-59593-577-0. DOI: 10.1145/1215995.1216004. URL: http://doi.acm.org/10.1145/1215995.1216004 (cit. on pp. 1, 2, 12).

[42] Brian Cantwell Smith. “Reflection and Semantics in LISP”. In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’84. Salt Lake City, Utah, USA: ACM, 1984, pp. 23–35. ISBN: 0-89791-125-3. DOI: 10.1145/800017.800513. URL: http://doi.acm.org/10.1145/800017.800513 (cit. on p. 3).

[43] Patrick G Sobalvarro. “A lifetime-based garbage collector for LISP systems on general-purpose computers”. PhD thesis. Massachusetts Institute of Technology, 1988 (cit. on p. 8).

[44] Michael Sperber et al. “Revised^6 report on the algorithmic language Scheme”. In: *Journal of Functional Programming* 19.S1 (2009), pp. 1–301 (cit. on pp. 2, 3, 7).

[45] Gerry Sussman, Harold Abelson, and Julie Sussman. *Structure and interpretation of computer programs*. MIT Press, Cambridge, Mass, 1983 (cit. on pp. 2, 3, 12).

[46] S Doaitse Swierstra. “Combinator parsers: From toys to tools”. In: *Electronic Notes in Theoretical Computer Science* 41.1 (2001), pp. 38–59 (cit. on p. 9).

[47] Don Syme, Gregory Neverov, and James Margetson. “Extensible pattern matching via a lightweight language extension”. In: *ACM SIGPLAN Notices*. Vol. 42. 9. ACM. 2007, pp. 29–40 (cit. on p. 7).

[48] David A. Terei and Manuel M.T. Chakravarty. “An LLVM Backend for GHC”. In: *Proceedings of the Third ACM Haskell Symposium on Haskell*. Haskell ’10. Baltimore, Maryland, USA: ACM, 2010, pp. 109–120. ISBN: 978-1-4503-0252-4. DOI: 10.1145/1863523.1863538. URL: http://doi.acm.org/10.1145/1863523.1863538 (cit. on pp. 8, 9).

[49] Luke VanderHart and Stuart Sierra. “Macros and Metaprogramming”. In: *Practical Clojure*. Springer, 2010, pp. 167–178 (cit. on p. 2).

[50] John Von Neumann. “First Draft of a Report on the EDVAC”. In: *IEEE Annals of the History of Computing* 15.4 (1993), pp. 27–75 (cit. on p. 3).

[51] David S. Wise. “Functional Programming”. In: *Encyclopedia of Computer Science*. Chichester, UK: John Wiley and Sons Ltd., pp. 736–739. ISBN: 0-470-86412-5. URL: http://dl.acm.org/citation.cfm?id=1074100.1074416 (cit. on p. 2).