Technical Report CS2016-12

Mnemosyne: A Functional Systems Programming Language

Hawk Weisman

Submitted to the Faculty of
The Department of Computer Science

Project Director: Dr. Robert Roos
Second Reader: John Wenskovitch

Allegheny College
2015

I hereby recognize and pledge to fulfill my responsibilities as defined in the Honor Code, and to maintain the integrity of both myself and the college community as a whole.

Hawk Weisman

HAWK WEISMAN. Mnemosyne: A Functional Systems Programming Language. (Under the direction of Dr. Robert Roos.)

Abstract

While programming languages researchers have produced a number of languages with various qualities that promote effective programming, such languages are frequently inappropriate for low-level systems programming. The automation of memory management through garbage collection is a major factor limiting the use of many popular languages for systems programming tasks.

Mnemosyne is a new functional programming language intended for systems programming, providing methods of ensuring memory safety at compile-time rather than at runtime. This report outlines the design and development of a prototype Mnemosyne implementation and provides an abbreviated specification for the language.

Contents

Chapter 1

Introduction

> For last year's words belong to last year's language
> And next year's words await another voice.
>
> *Four Quartets*
> T.S. ELIOT

The design and implementation of programming languages has been a major area of study for most of the history of computer science as a discipline. Since only a small fraction of software is implemented in assembly language, it seems reasonable to surmise that the quality of programming languages has a major impact on the quality of software in general. While many have observed that a great programmer can write good code even in a bad language[1], we should note that individuals with such wizardly skill[2] likely fall far outside of the bell curve distribution of programmer ability. For a majority of software developers, programming languages likely have at least some impact on software quality.

Given this observation, the programming languages community has, in recent years, produced a number of new programming languages, such as Haskell [29, 35], Scala [52, 53], 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, while the developers of application software have benefited significantly from recent advances in programming languages, the languages used in systems programming have changed little since the 1970s.

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 [50].

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 [50, 58]. This is in contrast

[1] And inversely, that a terrible programmer can write bad code even in a good language
[2] Or frightening incompetence

to *application programming*, which is concerned with the implementation of application software, software that the user interacts with directly or uses to accomplish a task or activity.

1.1 Current State of the Art

The *lingua franca* of systems programming is the C programming language [36], which first appeared in 1972. While there have been revisions to the C standard in recent years [33, 34] and new C compilers, such as `clang` [39], have been developed, C has changed very little since its creation. C is plagued by safety issues and can be very difficult to program in, especially for novice programmers or those unfamiliar with its challenges [5, 55, 58].

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 [58]. 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.

Many modern programming languages exhibit characteristics not suitable for systems programming. Many require a heavyweight runtime environment, running in a virtual machine or interpreter, making them insufficiently "close to the machine". They may provide abstractions and semantics that are convenient for the programmer but which make the low-level control necessary for many systems tasks impossible. When faced with the tradeoff between programmer convenience and low-level access to the computer's hardware, the designers of these languages tend, not unreasonably, to choose the former. While this choice has benefitted the implementors of computer applications greatly, it has also left systems programmers stuck with the relics of computing's past.

A particularly major issue, discussed in greater detail in Section 2.1, is that a vast majority of these languages manage memory automatically through garbage collection [4, 13]. While garbage collection means that programmers no longer have to worry about managing memory allocation and deallocation, it also means that this work must be performed at runtime. This interrupts the program's execution — often unsuitable for systems software – and requires runtime support from the garbage collection software [4, 13, 25].

1.2 Goals of the Project

Mnemosyne[3] is a new programming language intended for systems programming. Mnemosyne is intended to reconcile the requirements of low-level systems implementation with the

[3] Named for the Titaness who personified the concept of memory in Greek myth.

safety, expressive power, and elegance of modern functional programming languages. Mnemosyne is a compiled language with strong, static typing, Lisp-inspired S-expression syntax, and a focus on safe low-level programming. The most important of Mnemosyne's defining characteristics is a focus on automatic compile-time memory management, which should make it suitable for implementing systems software without necessitating programmers to allocate memory manually.

The primary goal of this project is to implement a working prototype of the Mnemosyne compiler, and to demonstrate the language's viability. Additionally, an abbreviated specification has been developed, and design considerations for currently unimplemented functionality are also discussed.

1.3 Thesis Outline

Chapter 2 provides greater detail into the rationale behind Mnemosyne's creation and reviews preceding work in the area. Chapter 3 discusses the considerations involved in the design of Mnemosyne's syntax and semantics, while Chapter 4 describes the implementation process for the prototype Mnemosyne compiler. In Chapter 5, methods of evaluating the correctness of the Mnemosyne compiler prototype are discussed, and the results of these evalautions analyzed. Finally, Chapter 6 concludes this thesis with a summation of the outcomes of the research and a discussion of potential avenues for future work.

Chapter 2

Background and Rationale

> C makes it easy to shoot yourself in the foot.
>
> PROGRAMMER'S APHORISM

2.1 Memory Management

Chapter 3

Design Considerations

A language that doesn't affect the way you think about programming is not worth knowing.

Epigrams on Programming [54]
ALAN PERLIS

3.1 Design Goals

As is true for all software systems, when implementing a new programming language, it is generally advantageous to identify a set of primary goals before beginning the implementation process.

The following characteristics are major design goals for Mnemosyne:

1. Support for systems programming
2. Safety
3. Performance
4. Expressiveness

Support for systems programming As Mnemosyne is intended as a language for systems programming, it stands to reason that the first object of consideration in all our design choices is to ensure that the language is suitable for this purpose. Thus, we might wonder, what characteristics or traits of a language make it suitable for systems programming?

There are some low-hanging fruit here, some language characteristics that seem immediately obvious. Of course, a systems language must be compiled rather than interpreted; many systems programs must run with little or no runtime support from other software, so an interpreter or just-in-time compiler is out of the question.

In "Programming Language Challenges in Systems Codes: Why Systems Programmers Still Use C, and What to Do About It", Shapiro enumerates quite well a number of important qualities of systems software. Shapiro observes that performance is much more important to the success of systems software than it is to applications, necessitating high-performance ways of representing data; and that bulk input and output has a large impact on the performance of such software. He states that systems programs often "operate in constrained memory", which he observes to have "unpleasant" implications for garbage collection. [58, pp. 2] Finally, he also points out that systems programs retain a great deal of state over the course of their execution, which "penalizes the performance of automatic storage reclamation strategies[1]" [58, pp. 2], as well as rendering unworkable the enforced purity or statelessness of functional languages such as Haskell.

Shapiro is not the only writer to highlight that garbage-collected languages are generally unsuitable for low-level systems implementation. 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 [25] 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.

In addition to eschewing the use of garbage collection, we must provide programmers with low-level access to hardware in order to support systems programming. In "Demystifying Magic: High-level Low-level Programming", Frampton et al. suggest the following fundamental requirements for systems languages: they must provide means of introducing new unboxed types and new semantics into the language, and they must provide mechanisms of *bypassing abstractions* when necessary [18]. We should wish to create abstractions that make programming simple and easy, then, but ensure that an "escape hatch" is always present when a lower level of abstraction is needed. For example, if our language passes references as typed smart pointers, we must ensure that it is possible to convert a reference into a raw pointer type if we need to perform pointer arithmetic or access individual bytes of memory — functionality that the implementers of an operating system kernel may require.

Safety Secondary only to our desire to support low-level systems programming, we should strive to ensure that Mnemosyne provides programmers with the tools to write code that is as safe as possible. At first blush, our first and second goals seem natural enemies — by allowing programmers a way out of language abstractions or providing unrestricted access to hardware, we are allowing new classes of errors and making our programs less safe. However, with thought, we can find ways to reconcile these competing needs.

Frampton et al. discuss the importance of what they call "containment": the idea that

[1] *Viz.* garbage collection

safe code should not be tainted by the unsafe operations sometimes sadly necessary for systems-level code, as discussed in the previous paragraph. Mechanisms of containment, according to Frampton et al., work to “minimiz[e] the reach of unsafe operations and maximiz[e] the scope of untainted high-level code” [18]. The Rust programming language employs an approach similar to that discussed by Frampton et al.: certain operations which the compiler cannot reason about but are nonetheless sometimes necessary are designated as ‘unsafe’, and may only be used within functions and blocks of code designated as unsafe by the programmer. These unsafe operations include dereferencing raw pointers[2], a violation of the language’s memory-management system; non-scalar casts[3], which violate the type system, and the use of any other function designated as unsafe by its implementor [6, 46]. If issues arising from the use of such unsafe operations occur, they are thus confined to those code regions designated as unsafe, making the task of finding the source of these issues much easier.

Memory safety is a major issue in C and other languages which employ manual memory management. Manual memory management results in a number of error categories, such as dangling pointers, buffer overflows, and null pointer derefences, that manifest themselves frequently in programs written in languages without garbage collection [5, 22, 55, 58]. Mnemosyne’s automatic compile-time memory management eliminates these errors, providing the same safety advantages as garbage collection without rendering the language useless for systems software.

Performance The performance of a programming language is primarily a result of how programs written in that language are executed; that is to say, the compiler or interpreter. As stated in Section 3.2, Mnemosyne is a compiled language rather than an interpreted language, and compiled languages almost universally offer better performance than interpreted ones. However, we must take care to ensure that the compiler outputs high-performance binaries.

LLVM, the compiler backend used by Mnemosyne, has been noted for its performance [39, 40, 67]. LLVM is capable of performing a number of optimizations on its internal intermediate representation while generating output binaries, and Mnemosyne programs will benefit from these optimizations in addition to those performed by the compiler at earlier stages in the compilation process.

Furthermore, performance can be improved by providing *zero-cost abstractions*. Zero-cost abstractions are those which do not incur an additional performance cost, either in time or in space, at runtime. Essentially, thse are abstractions which are erased by the compiler as part of the compilation process rather than impacting the output binaries of a program. By focusing on abstractions that are zero-cost when designing our language semantics and standard library, we can improve performance while still providing programmers with an expressive set of tools for writing elegant, reusable, and concise code.

[2] Those which are not part of Rust’s lifetime-analysis system.

[3] Called ‘transmutation’.

Expressiveness Formally, the *expressiveness* (or *expressive power*) refers to the size of the set of all ideas or concepts capable of being communicated by a language. As a general-purpose programming language, Mnemosyne is intended to be Turing-complete; meaning that theoretically, all possible computations can be expressed in Mnemosyne.

In the context of general-purpose, Turing-complete languages, this term is more often used to refer to the *ease* with which complex concepts can be expressed: a more expressive language is one wherein a complex idea requires less code. We could, then, perhaps informally define this kind of expressiveness as a ratio of things accomplished to lines of code. While this definition does not provide us with a measurable metric — "things accomplished" is a bit too nebulous – it neatly summarizes the idea that we should like for programmers to be able to accomplish a great deal in as few lines of code as possible.

Ensuring that Mnemosyne is as expressive as possible primarily involves the design of the language's syntax, as discussed in Section 3.3. The S-expression notation used for Mnemoysne programs has been noted for its expressiveness [56, 64]. Standard library design also influences a language's expressiveness; by providing powerful abstractions in the standard library, we can eliminate 'boilerplate' code and permit our programmers to do more with less.

3.2 Characteristics

Programming languages may be categorized along a number of axes: whether it is compiled or interpreted, its typing discipline, what programming paradigms it is inteded to support, its method of memory management, and others.

Mnemosyne is a *statically-typed* programming language, meaning that the analysis of the types of language constructs, such as variables and expressions, are known to the compiler at compile-time. This is in contrast to *dynamically-typed* languages, such as a majority of the Lisp family, in which types are determined at runtime [48].

A major advantage of static typing lies in its increased reliability. If we suppose that some operations are not supported on all types, and that attempting these operations on types which are not capable of carrying them out results in an error, as is the case in almost all programming languages, then such *type errors* are a category of potential error which may occur in our programs. In a statically-typed language, the compiler has the capacity to reason about types, and thus these type errors can be detected at compile-time; while in a dynamically typed language, type errors will occur during the program's execution. Moving the detection of an entire category of errors from runtime to compile-time is a major boon to the language's reliability and safety [47, 48].

Furthermore, many programming languages use their type systems to encode other potential categories of errors in their type systems. The use of types in this manner can be used to detect security issues [59] and concurrency defects [57] statically (i.e., at compile-time), rather than allowing these errors to remain undetected in software deployed into production. Additionally, giving the compiler the capacity to reason about types permits us to perform a number of optimizations that are not possible in dynamically-typed languages.

In contrast, dynamic typing has frequently been observed to make writing amd modifying code less time-consuming [47]. This quality is generally very useful for scripting languages, rapid prototyping, and educational programming languages. These domains lie outside of Mnemosyne's primary design goals, and dynamic typing is generally unsuitable for systems programming.

Mnemosyne is intended to support programming in the *functional* paradigm.

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 [30, 71].

Functional programming models the execution of a program as the evalaution of a series of functions. This paradigm of programming has been observed to be very expressive and to encourage safe programming practices, primarily based on the control of side effects [28, 30]. Furthermore, functional programming has also been noted to enable highly modular architectural and design practices, resulting in software that is easier to test and code that can be reused often [28, 30].

3.3 Syntax

The *syntax* of a programming language refers to its lexical and grammatical stucture; the combinations of characters and strings that make up valid programs in that language. [23] Due to the obvious impact of these considerations on the programs written in a language, the syntactical design of a programming langauge must be approached with some care. While appendix A furnishes a formal description of the Mnemosyne grammar, this section discusses the rationale behind the design decisions of the language's syntax.

Mnemosyne borrows the *S-expression*-based syntax of the Lisp family of languages.

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.

S-expression syntax has a number of advantages. Perhaps the most important property of S-expressionsis 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 [64, 68]. In Mnemosyne, as 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.

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.

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 [54]."

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 [27, 29, 35]. Pattern matching is a syntactic construct common to functional programming languages such as Haskell [27, 29, 35], Scala [52, 53], and ML [38, 43]. 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 [38, 43, 44, 66] 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 [29, 35].

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 [27]. 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[4], 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 [46].

3.3.0.1 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 [62] (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 [2, 3] and Hawblitzel et al. [22], 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'.

Definition 4 (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 [10, 21].

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 [10, 21]. 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

[4] As in Haskell.

created, placing significant constraints on the programmer and limiting the expressiveness of the language.

In order to make such a language expressive enough to be useable, a Rust-like system of ownership and lending seems necessary. 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 [6, 46]. 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. Previous research [61] suggests that lifetime analysis for Lisp programs is certainly possible, lending credibility to the use of this approach in Mmenosyne.

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 [8, 14, 15], to the extent that the originator of the `null` reference, Sir C.A.R. Hoare, referred to it as his "billion-dollar mistake" [26].

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 [15]. 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 [15]. A number of popular functional programming languages provide such a type: Scala and Rust call it `Option` [46, 52, 53] while Haskell calls it `Maybe` [27, 35]. 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.

Chapter 4

Implementation

The Mnemosyne compiler, called `mn`[1] operates in three primary phases:

- **Semantic analysis** converts the program source code to a representation understandable by the compiler and detects syntax errors
- **Semantic analysis** attempts to prove statements about the program's execution, such as determining the types of values and reducing expressions to constants, and detects semantic errors
- **Code generation** converts the internal representation of the program to LLVM intermediate representation (IR) and then to the desired output binary format.

4.1 Parsing and Syntactic Analysis

The Mnemosyne parser is implemented using technique called *combinator parsing*.

4.2 Semantic Analysis

4.3 Code Generation

[1]Pronounced "Manganese".

Chapter 5

Evaluation

Another possible chapter title: Experimental Results

Chapter 6

Discussion and Future Work

The future will be better tomorrow.

DAN QUAYLE

This chapter usually contains the following items, although not necessarily in this order or sectioned this way in particular.

6.1 Summary of Results

A discussion of the significance of the results and a review of claims and contributions.

6.2 Future Work

6.3 Conclusion

Appendix A

The Mnemosyne Programming Language: An Abridged Description

A.1 Introduction

> I really do not know that anything has ever been more exciting than diagramming sentences.
>
> *Lectures in America*
> GERTRUDE STEIN

The following is an abbreviated description of the Mnemosyne programming language. It is for illustrative purposes only and is not intended as a complete formal specification.

Please note that as the Mnemosyne compiler is currently an early prototype, it may not behave exactly as described in this document. While the compiler remains in the prototype stage (i.e., prior to the release of version 1.0.0), please regard this document as the only description of the canonical behaviour of a standards-compliant Mnemosyne compiler.

A.1.1 Syntactic Notation

Syntax descriptions are written using an extended BNF notation, as follows:

$\langle symbol \rangle$ indicates a non-terminal symbol

'`symbol`' indicates a terminal symbol

$\langle symbol \rangle$* indicates zero or more repetitions of $\langle symbol \rangle$

$\langle symbol \rangle$+ indicates one or more repetitions of $\langle symbol \rangle$.

ε indicates the empty string

The following special symbols refer to specific Unicode characters:

$\langle lambda \rangle$ Greek capital letter Lambda (U+03BB)

$\langle$*arrow*$\rangle$ Rightwards arrow (U+2192)

$\langle$*double arrow*$\rangle$ Rightwards double arrow (U+21D2)

$\langle$*tab*$\rangle$ Character tabulation (U+0009)

$\langle$*linefeed*$\rangle$ Linefeed (U+000A)

$\langle$*return*$\rangle$ Carriage return (U+000D)

$\langle$*space*$\rangle$ Space (U+0020)

Finally, the symbol $\langle$*any*$\rangle$ refers to any character.

A.2 Program Structure

All Mnemosyne programs consist of one or more *modules*. A module forms the top level of a Mnemosyne program and represents a namespace within which types and functions may be defined. A module then consists of a series of one or more *definitions* (Appendix A.5) and *expressions* (Appendix A.4). An expression is a value-level construct: all expressions can be evaluated to some value, either at run-time or at compile-time. Definitions, by contrast, are type-level constructs.

A.3 Lexical Syntax

This section describes the lexical structure of Mnemosyne programs.

Mnemosyne uses the Unicode character set. While Mnemosyne programs written in the ASCII character set are considered valid, a standards-compiliant Mnemosyne compiler should be capable of recognizing Unicode characters. Unicode support is necessary both because certain non-alphanumeric Unicode characters not present in ASCII have defined meanings in Mnemosyne programs, and in order to ensure that Mnemosyne programs may be written in languages other than English. Note that Mnemosyne's lexical syntax then depends on the properties of the Unicode encoding as defined by the Unicode consortium. A standards-compilant Mnemosyne compiler should ensure compatiblity with new versions of the Unicode standard as they are released.

Note: **??** contains a source code listing for the Mnemosyne parser, and should be referred to to answer specific questions regarding how the reference Mnemosyne implementation handles specific characters.

$\langle$*program*$\rangle$ $\rightarrow$ $\langle$*token*$\rangle$+

$\langle$*token*$\rangle$ $\rightarrow$ $\langle$*lexeme*$\rangle$ | $\langle$*atmosphere*$\rangle$

$\langle$*lexeme*$\rangle$ $\rightarrow$ $\langle$*identifier*$\rangle$ | $\langle$*operator*$\rangle$ | $\langle$*keyword*$\rangle$ | $\langle$*literal*$\rangle$
| $\langle$*sigil*$\rangle$ | $\langle$*delimiter*$\rangle$

⟨*sigil*⟩ → ‘`@`’ | ‘`&`’ | ‘`*`’ | ‘`$`’ | ‘`?`’

⟨*delimiter*⟩ → ‘`(`’ | ‘`)`’ | ‘`{`’ | ‘`}`’

⟨*identifier*⟩ → ⟨*initial*⟩ ⟨*subsequent*⟩*

⟨*initial*⟩ → ⟨*letter*⟩ | ⟨*special initial*⟩

⟨*subsequent*⟩ → ⟨*letter*⟩ | ⟨*number*⟩ | ⟨*special subsequent*⟩

⟨*letter*⟩ → ‘`a`’ | ‘`b`’ | ‘`c`’ | ... | ‘`z`’
| ‘`A`’ | ‘`B`’ | ‘`C`’ | ... | ‘`Z`’

⟨*number*⟩ → ‘`0`’ | ‘`1`’ | ... | ‘`9`’

⟨*special initial*⟩ → ‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`<`’ | ‘`>`’ | ‘`=`’
| ‘`!`’ | ‘`:`’ | ‘`%`’ | ‘`^`’

⟨*special subsequent*⟩ → ⟨*special initial*⟩ | ‘`’`’ | ‘`_`’

⟨*keyword*⟩ → ‘`and`’ | ‘`begin`’ | ‘`borrow`’ | ‘`case`’ | ‘`cond`’
| ‘`class`’ | ‘`data`’ | ‘`define`’ | ‘`defn`’ | ‘`def`’
| ‘`delay`’| ‘`do`’ | ‘`else`’ | ‘`if`’ | ‘`instance`’
| ‘`impl`’ | ‘`lambda`’ | ‘`let`’ | ‘`let*`’ | ‘`letrec`’
| ‘`mod`’ | ‘`or`’ | ‘`quasiquote`’ | ‘`quote`’ | ‘`ref`’
| ‘`set!`’ | ‘`struct`’ | ‘`trait`’ | ‘`type`’ | ‘`typeclass`’
| ‘`union`’ | ‘`unquote`’ | ‘`unquote-splicing`’
| ⟨*lambda*⟩ | ⟨*arrow*⟩ | ⟨*fat arrow*⟩
| ⟨*builtin type*⟩
| ‘`|`’ | ‘`->`’ | ‘`=>‘ | ‘,`’

⟨*builtin type*⟩ → ‘`i8`’ | ‘`i16`’ | ‘`i32`’ | ‘`i64`’ | ‘`int`’
| ‘`u8`’ | ‘`ui6‘ | ‘u32`’ | ‘`u64`’ | ‘`uint`’
| ‘`f32`’ | ‘`f64`’ | ‘`float`’ | ‘`double`’
| ‘`bool`’ | ‘`string`’

⟨*atmosphere*⟩ → ⟨*whitespace*⟩ | ⟨*comment*⟩

⟨*whitespace*⟩ → ⟨*space*⟩
| ⟨*tab*⟩ (U+0009)
| ⟨*linefeed*⟩ (U+000A)
| ⟨*carriage return*⟩ (U+000D)

⟨*comment*⟩ → ‘`;`’ ⟨*any*⟩* ⟨*line ending*⟩
| ‘`#|`’ ⟨*any*⟩* ‘`|#`’

A.4 Expressions

Mnemosyne *expressions* form the basic building block from which all Mnemosyne programs are constructed. An expression is defined as any sequence of Mnemosyne tokens which may be resolved to a value-level result, either by the compiler or during the execution of a program. This section describes the syntax and informal semantics of Mnemosyne expressions.

$$
\begin{array}{lcl}
\langle \textit{expr} \rangle & \rightarrow & \langle \textit{s-expr} \rangle \mid \langle \textit{i-expr} \rangle \mid \langle \textit{c-expr} \rangle \mid \langle \textit{n-expr} \rangle \\
& \mid & \langle \textit{deref expr} \rangle \mid \langle \textit{unwrap expr} \rangle \mid \langle \textit{pointer expr} \rangle \\
& \mid & \langle \textit{literal} \rangle
\end{array}
$$

$$\langle \textit{s-expr} \rangle \rightarrow \text{‘}\texttt{(}\text{’}\ \langle \textit{operator} \rangle\ \langle \textit{expr} \rangle^{*}\ \text{‘}\texttt{)}\text{’}$$

$$\langle \textit{c-expr} \rangle \rightarrow \text{‘}\texttt{\{}\text{’}\ \langle \textit{expr} \rangle\ \langle \textit{operator} \rangle\ \langle \textit{c-expr body} \rangle+\ \text{‘}\texttt{\}}\text{’}$$

$$
\begin{array}{lcl}
\langle \textit{c-expr body} \rangle & \rightarrow & \langle \textit{expr} \rangle\ \langle \textit{operator} \rangle \\
& \mid & \langle \textit{expr} \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle \textit{n-expr} \rangle & \rightarrow & \langle \textit{access} \rangle\ \text{‘}\texttt{(}\text{’}\ \langle \textit{expr} \rangle^{*}\ \text{‘}\texttt{)}\text{’} \\
& \mid & \langle \textit{access} \rangle\ \text{‘}\texttt{.}\text{’}\ \langle \textit{identifier} \rangle \\
& \mid & \langle \textit{access} \rangle\ \text{‘}\texttt{.}\text{’}\ \langle \textit{n-expr} \rangle
\end{array}
$$

$$\langle \textit{access} \rangle \rightarrow \langle \textit{deref-expr} \rangle \mid \langle \textit{identifier} \rangle$$

A.4.1 Dereference and Unwrapping Expressions

$$
\begin{array}{lcl}
\langle \textit{deref expr} \rangle & \rightarrow & \text{‘}\texttt{(}\text{’}\ \text{‘}\texttt{\$}\text{’}\ \langle \textit{expr} \rangle\ \text{‘}\texttt{)}\text{’} \\
& \mid & \text{‘}\texttt{\$}\text{’}\ \langle \textit{expr} \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle \textit{unwrap expr} \rangle & \rightarrow & \text{‘}\texttt{(}\text{’}\ \text{‘}\texttt{?}\text{’}\ \langle \textit{expr} \rangle\ \langle \textit{expr} \rangle \text{‘}\texttt{)}\text{’} \\
& \mid & \text{‘}\texttt{(}\text{’}\ \text{‘}\texttt{?}\text{’}\ \langle \textit{expr} \rangle\ \text{‘}\texttt{)}\text{’} \\
& \mid & \text{‘}\texttt{?}\text{’} \langle \textit{expr} \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle \textit{pointer expr} \rangle & \rightarrow & \text{‘}\texttt{(}\text{’}\ \langle \textit{pointer sigil} \rangle\ \langle \textit{expr} \rangle\ \text{‘}\texttt{)}\text{’} \\
& \mid & \langle \textit{pointer sigil} \rangle\ \langle \textit{expr} \rangle
\end{array}
$$

$$\langle \textit{pointer sigil} \rangle \rightarrow \text{‘}\texttt{\&}\text{’} \mid \text{‘}\texttt{@}\text{’} \mid \text{‘}\texttt{*}\text{’}$$

A.4.2 Modules

$$\langle \textit{program} \rangle \rightarrow \langle \textit{module} \rangle+$$

$$\langle \textit{module} \rangle \rightarrow \langle \textit{module-def} \rangle\ \langle \textit{definition} \rangle^{*}$$

$$\langle \textit{module-def} \rangle \rightarrow \text{‘}\texttt{(}\text{’}\ \text{‘}\texttt{mod}\text{’}\ \langle \textit{identifier} \rangle\ \langle \textit{exports-clause} \rangle\ \text{‘}\texttt{)}\text{’}$$

$\langle \textit{exports-clause} \rangle \rightarrow$ ‘`(`’ ‘`exports`’ $\langle \textit{identifier} \rangle$+ ‘`)`’
 | ε

A.5 Definitions

A.5.1 Attributes

$\langle \textit{function-attrs} \rangle \rightarrow$ ‘`#(`’ $\langle \textit{function-attr} \rangle$* ‘`)`’

$\langle \textit{function-attr} \rangle \rightarrow \langle \textit{any-attr} \rangle$
 | ‘`cold`’
 | $\langle \textit{inline} \rangle$

$\langle \textit{inline} \rangle \rightarrow$ ‘`inline`’
 | ‘`inline-always`’

$\langle \textit{data-attr} \rangle \rightarrow \langle \textit{any-attr} \rangle$
 | ‘`(`’ ‘`as`’ $\langle \textit{as-attr} \rangle$+ ‘`)`’
 | ‘`as`’ ‘`(`’ $\langle \textit{as-attr} \rangle$+ ‘`)`’

$\langle \textit{as-attr} \rangle \rightarrow$ ‘`C`’ | ‘`packed`’
 | ‘`u8`’ | ‘`u16`’ | ‘`u32`’ | ‘`u64`’

Appendix B

Manganese Source Code

> Accursed creator! Why did you form a monster so hideous that even you turned from me in disgust?
>
> *Frankenstein*
> MARY SHELLEY

B.1 Mnemosyne Core Crate

lib.rs

```
//
// Mnemosyne: a functional systems programming language.
// (c) 2015 Hawk Weisman
//
// Mnemosyne is released under the MIT License. Please refer to
// the LICENSE file at the top-level directory of this distribution
// or at https://github.com/hawkw/mnemosyne/.
//
#![crate_name = "mnemosyne"]
#![crate_type = "lib"]
#![feature(rustc_private)]
#![feature(static_recursion)]
#![feature(box_syntax, box_patterns)]

//! # Mnemosyne core
//!
//! This crate contains the core Mnemosyne programming language components.
//! This includes the mnemosyne abstract syntax tree (`semantic::ast`),
//! functions for performing semantic analysis (`semantic`), functions
//! for compiling abstract syntax trees to LLVM bytecode (`compile`), and
//! assorted utility code such as a positional reference type and a
//! `ForkTable` data structure for use as a symbol table.
//!
//! The Mnemosyne parser is contained in a separate crate in order to improve
```

```
//! compile times.

extern crate rustc;
extern crate libc;
extern crate combine;
extern crate iron_llvm;
extern crate llvm_sys;
#[macro_use] extern crate itertools;

use rustc::lib::llvm::{LLVMVersionMajor, LLVMVersionMinor};

use std::fmt::Debug;

include!(concat!(env!("OUT_DIR"), "/gen.rs"));

/// Returns the Mnemosyne version as a String
pub fn mnemosyne_version() -> String {
    format!("Mnemosyne {}", env!("CARGO_PKG_VERSION"))
}

/// Macro for formatting an internal compiler error panic.
///
/// This should be used instead of the Rust standard library's `panic!()`
/// macro in the event of an unrecoverable internal compiler error.
#[macro_export]
macro_rules! ice {
    ($msg:expr) => (
        panic!( "[internal error] {}\n \
                 [internal error] Something has gone horribly wrong.\n \
                 [internal error] Please contact the Mnemosyne implementors.\n\
                 {}, {}"
               , $msg
               , $crate::mnemosyne_version(), $crate::llvm_version()
               )
          );
    ($fmt:expr, $($arg:tt)+) => (
        panic!( "[internal error] {}\n \
                 [internal error] Something has gone horribly wrong.\n \
                 [internal error] Please contact the Mnemosyne implementors.\n\
                 {}, {}"
               , format_args!($fmt, $($arg)+)
               , $crate::mnemosyne_version(), $crate::llvm_version()
               )
          )
}

pub mod position;
pub mod semantic;
pub mod compile;
pub mod forktable;
pub mod chars;
pub mod errors;
```

```
pub use semantic::ast;
```

chars.rs

```
/// Unicode code point for the lambda character
pub const LAMBDA: &'static str       = "\u{03bb}";
/// Unicode code point for the arrow character
pub const ARROW: &'static str        = "\u{8594}";
/// Unicode code point for the fat arrow (typeclass) character.
pub const FAT_ARROW: &'static str    = "\u{8685}";

pub const ALPHA_EXT: &'static str    = "+-*/<=>!:$%_^";
pub const OPS: &'static str          = "+-*/|=<>";
```

errors.rs

```
//
// Mnemosyne: a functional systems programming language.
// (c) 2015 Hawk Weisman
//
// Mnemosyne is released under the MIT License. Please refer to
// the LICENSE file at the top-level directory of this distribution
// or at https://github.com/hawkw/mnemosyne/.
//

use std::fmt::{ Display, Debug };

/// Mnemosyne error handling

/// Wraps Option/Result with an ‘expect_ice()‘ method.
///
/// The ‘expect_ice()‘ method functions similarly to the standard library's
/// ‘expect()‘, but with the custom Mnemosyne internal compiler error message.
pub trait ExpectICE<T> {
    fn expect_ice(self, msg: &str) -> T;
}

impl<T> ExpectICE<T> for Option<T> {
    /// Unwraps an option, yielding the content of a ‘Some‘
    ///
    /// # Panics
    ///
    /// Panics using the Mnemosyne internal compiler error formatter
    /// if the value is a ‘None‘, with a custom panic message
    /// provided by ‘msg‘.
    ///
    /// # Examples
    ///
    /// ‘‘‘ignore
    /// # use mnemosyne::errors::ExpectICE;
    /// let x = Some("value");
```

~~~~
    /// assert_eq!(x.expect_ice("the world is ending"), "value");
    /// ```
    ///
    /// ```ignore
    /// # use mnemosyne::errors::ExpectICE;
    /// let x: Option<&str> = None;
    /// x.expect_ice("the world is ending");
    /// ```
    #[inline]
    fn expect_ice(self, msg: &str) -> T {
        match self { Some(thing) => thing
                   , None        => ice!(msg)
                   }
    }
}

impl<T, E> ExpectICE<T> for Result<T, E>
where E: Debug {

    /// Unwraps a result, yielding the content of an `Ok`.
    ///
    /// Panics using the Mnemosyne internal compiler error formatter
    /// if the value is an `Err`, with a panic message including the
    /// passed message, and the content of the `Err`.
    ///
    /// # Examples
    /// ```ignore
    /// # use mnemosyne::errors::ExpectICE;
    /// let x: Result<u32, &str> = Err("emergency failure");
    /// x.expect_ice("Testing expect");
    /// ```
    #[inline]
    fn expect_ice(self, msg: &str) -> T {
        match self { Ok(t) => t
                   , Err(e) => ice!("{}: {:?}", msg, e)
                   }
    }
}

/// Wraps Option/Result with an `unwrap_ice()` method.
///
/// The `unwrap_ice()` method functions similarly to the standard library's
/// `unwrap()`, but with the custom Mnemosyne internal compiler error message.
pub trait UnwrapICE<T> {
    fn unwrap_ice(self) -> T;
}

impl<T> UnwrapICE<T> for Option<T> {
    /// Moves the value `v` out of the `Option<T>` if it is `Some(v)`.
    ///
    /// Unlike the standard library's `unwrap()`, this uses the Mnemosyne
    /// internal compiler error panic formatter.
~~~~

```
    ///
    /// # Panics
    ///
    /// Panics if the self value equals `None`.
    ///
    /// # Safety note
    ///
    /// In general, because this function may panic, its use is discouraged.
    /// Instead, prefer to use pattern matching and handle the `None`
    /// case explicitly.
    ///
    /// # Examples
    ///
    /// ```ignore
    /// # use mnemosyne::errors::UnwrapICE;
    /// let x = Some("air");
    /// assert_eq!(x.unwrap_ice(), "air");
    /// ```
    ///
    /// ```ignore
    /// # use mnemosyne::errors::UnwrapICE;
    /// let x: Option<&str> = None;
    /// assert_eq!(x.unwrap_ice(), "air"); // fails
    /// ```
    #[inline]
    fn unwrap_ice(self) -> T {
        match self { Some(thing) => thing
                   , None =>
                        ice!("called `Option::unwrap()` on a `None` value")
                   }
    }
}

impl<T, E> UnwrapICE<T> for Result<T, E>
where E: Display  {
    /// Unwraps a result, yielding the content of an `Ok`.
    ///
    /// Unlike the standard library's `unwrap()`, this uses the Mnemosyne
    /// internal compiler error panic formatter.
    ///
    /// # Panics
    ///
    /// Panics if the value is an `Err`, with a panic message provided by the
    /// `Err`'s value.
    ///
    /// # Examples
    ///
    /// ```ignore
    /// # use mnemosyne::errors::UnwrapICE;
    /// let x: Result<u32, &str> = Ok(2);
    /// assert_eq!(x.unwrap_ice(), 2);
    /// ```
```

```
    ///
    /// ```ignore
    /// # use mnemosyne::errors::UnwrapICE;
    /// let x: Result<u32, &str> = Err("emergency failure");
    /// x.unwrap_ice(); // panics
    /// ```
    #[inline]
    fn unwrap_ice(self) -> T {
        match self { Ok(t) => t
                   , Err(e) => ice!("{}", e)
                   }
    }
}
//
// impl<T, E> UnwrapICE<T> for Result<T, E>
// where E: Debug {
//     /// Unwraps a result, yielding the content of an `Ok`.
//     ///
//     /// Unlike the standard library's `unwrap()`, this uses the Mnemosyne
//     /// internal compiler error panic formatter.
//     ///
//     /// # Panics
//     ///
//     /// Panics if the value is an `Err`, with a panic message provided by the
//     /// `Err`'s value.
//     ///
//     /// # Examples
//     ///
//     /// ```
//     /// # use mnemosyne::errors::UnwrapICE;
//     /// let x: Result<u32, &str> = Ok(2);
//     /// assert_eq!(x.unwrap_ice(), 2);
//     /// ```
//     ///
//     /// ```{.should_panic}
//     /// # use mnemosyne::errors::UnwrapICE;
//     /// let x: Result<u32, &str> = Err("emergency failure");
//     /// x.unwrap_ice(); // panics with `emergency failure`
//     /// ```
//     #[inline]
//     fn unwrap_ice(self) -> T {
//         match self {
//             Ok(t) => t
//           , Err(e) =>
//                 ice!("called `Result::unwrap()` on an `Err` value: {:?}", e)
//         }
//     }
// }

#[cfg(test)]
mod tests {
    use super::*;
```

```
    #[test]
    fn test_option_expect_ok() {
        let x = Some("value");
        assert_eq!(x.expect_ice("the world is ending"), "value");
    }

    #[test]
    #[should_panic]
    fn test_option_expect_panic() {
        let x: Option<&str> = None;
        x.expect_ice("the world is ending");
    }

    #[test]
    #[should_panic]
    fn test_result_expect_panic() {
        let x: Result<u32, &str> = Err("emergency failure");
        x.expect_ice("Testing expect");
    }

    #[test]
    fn test_option_unwrap_ok() {
        let x = Some("air");
        assert_eq!(x.unwrap_ice(), "air");
    }

    #[test]
    #[should_panic]
    fn test_option_unwrap_panic() {
        let x: Option<&str> = None;
        assert_eq!(x.unwrap_ice(), "air"); // fails
    }

    #[test]
    fn test_result_unwrap_ok() {
        let x: Result<u32, &str> = Ok(2);
        assert_eq!(x.unwrap_ice(), 2);
    }

    #[test]
    #[should_panic]
    fn test_result_unwrap_panic() {
        let x: Result<u32, &str> = Err("emergency failure");
        x.unwrap_ice(); // panics
    }
}
```

forktable.rs

```
//
// Mnemosyne: a functional systems programming language.
```

```
// (c) 2015 Hawk Weisman
//
// Mnemosyne is released under the MIT License. Please refer to
// the LICENSE file at the top-level directory of this distribution
// or at https://github.com/hawkw/mnemosyne/.
//

use ::errors::ExpectICE;

use std::collections::{HashMap, HashSet};
use std::collections::hash_map::{Keys,Values};
use std::hash::Hash;
use std::borrow::Borrow;
use std::ops;

/// An associative map data structure for representing scopes.
///
/// A ‘ForkTable‘ functions similarly to a standard associative map
/// data structure (such as a ‘HashMap‘), but with the ability to
/// fork children off of each level of the map. If a key exists in any
/// of a child’s parents, the child will ’pass through’ that key. If a
/// new value is bound to a key in a child level, that child will overwrite
/// the previous entry with the new one, but the previous ‘key‘ -> ‘value‘
/// mapping will remain in the level it is defined. This means that the parent
/// level will still provide the previous value for that key.
///
/// This is an implementation of the ForkTable data structure for
/// representing scopes. The ForkTable was initially described by
/// Max Clive. This implemention is based primarily by the Scala
/// reference implementation written by Hawk Weisman for the Decaf
/// compiler, which is available [here](https://github.com/hawkw/decaf/blob/master/src/main/scal
#[derive(Debug, Clone)]
pub struct ForkTable<’a, K, V>
where K: Eq + Hash
    , K: ’a
    , V: ’a
{
    table: HashMap<K, V>
  , whiteouts: HashSet<K>
  , parent: Option<&’a ForkTable<’a, K, V>>
  , level: usize
}

impl<’a, K, V> ForkTable<’a, K, V>
where K: Eq + Hash
{

    /// Returns a reference to the value corresponding to the key.
    ///
    /// If the key is defined in this level of the table, or in any
    /// of its’ parents, a reference to the associated value will be
    /// returned.
```

```
    ///
    /// The key may be any borrowed form of the map’s key type, but
    /// ‘Hash‘ and ‘Eq‘ on the borrowed form *must* match those for
    /// the key type.
    ///
    /// # Arguments
    ///
    ///  + ‘key‘  - the key to search for
    ///
    /// # Return Value
    ///
    ///  + ‘Some(&V)‘ if an entry for the given key exists in the
    ///      table, or ‘None‘ if there is no entry for that key.
    ///
    /// # Examples
    ///
    /// ‘‘‘ignore
    /// # use mnemosyne::forktable::ForkTable;
    /// let mut table: ForkTable<isize,&str> = ForkTable::new();
    /// assert_eq!(table.get(&1), None);
    /// table.insert(1, "One");
    /// assert_eq!(table.get(&1), Some(&"One"));
    /// assert_eq!(table.get(&2), None);
    /// ‘‘‘
    /// ‘‘‘ignore
    /// # use mnemosyne::forktable::ForkTable;
    /// let mut level_1: ForkTable<isize,&str> = ForkTable::new();
    /// level_1.insert(1, "One");
    ///
    /// let mut level_2: ForkTable<isize,&str> = level_1.fork();
    /// assert_eq!(level_2.get(&1), Some(&"One"));
    /// ‘‘‘
    pub fn get<Q: ?Sized>(&self, key: &Q) -> Option<&V>
    where K: Borrow<Q>
        , Q: Hash + Eq
    {
        if self.whiteouts.contains(key) {
            None
        } else {
            self.table
                .get(key)
                .or(self.parent
                        .map_or(None, |ref parent| parent.get(key))
                    )
        }
    }

    /// Returns a mutable reference to the value corresponding to the key.
    ///
    /// If the key is defined in this level of the table, a reference to the
    /// associated value will be returned.
    ///
```

```
    /// Note that only keys defined in this level of the table can be accessed
    /// as mutable. This is because otherwise it would be necessary for each
    /// level of the table to hold a mutable reference to its parent.
    ///
    /// The key may be any borrowed form of the map's key type, but
    /// `Hash` and `Eq` on the borrowed form *must* match those for
    /// the key type.
    ///
    /// # Arguments
    ///
    ///  + `key`  - the key to search for
    ///
    /// # Return Value
    ///
    ///  + `Some(&mut V)` if an entry for the given key exists in the
    ///      table, or `None` if there is no entry for that key.
    ///
    /// # Examples
    ///
    /// ```ignore
    /// # use mnemosyne::forktable::ForkTable;
    /// let mut table: ForkTable<isize,&str> = ForkTable::new();
    /// assert_eq!(table.get_mut(&1), None);
    /// table.insert(1isize, "One");
    /// assert_eq!(table.get_mut(&1), Some(&mut "One"));
    /// assert_eq!(table.get_mut(&2), None);
    /// ```
    /// ```ignore
    /// # use mnemosyne::forktable::ForkTable;
    /// let mut level_1: ForkTable<isize,&str> = ForkTable::new();
    /// level_1.insert(1, "One");
    ///
    /// let mut level_2: ForkTable<isize,&str> = level_1.fork();
    /// assert_eq!(level_2.get_mut(&1), None);
    /// ```
    pub fn get_mut<Q: ?Sized>(&mut self, key: &Q) -> Option<&mut V>
    where K: Borrow<Q>
        , Q: Hash + Eq
    {
        self.table.get_mut(key)
    }


    /// Removes a key from the map, returning the value at the key if
    /// the key was previously in the map.
    ///
    /// If the removed value exists in a lower level of the table,
    /// it will be whited out at this level. This means that the entry
    /// will be 'removed' at this level and this table will not provide
    /// access to it, but the mapping will still exist in the level where
    /// it was defined. Note that the key will not be returned if it is
    /// defined in a lower level of the table.
```

```
    ///
    /// The key may be any borrowed form of the map's key type, but
    /// `Hash` and `Eq` on the borrowed form *must* match those for
    /// the key type.
    ///
    /// # Arguments
    ///
    ///  + `key`  - the key to remove
    ///
    /// # Return Value
    ///
    ///  + `Some(V)` if an entry for the given key exists in the
    ///     table, or `None` if there is no entry for that key.
    ///
    /// # Examples
    /// ```ignore
    /// # use mnemosyne::forktable::ForkTable;
    /// let mut table: ForkTable<isize,&str> = ForkTable::new();
    /// table.insert(1, "One");
    ///
    /// assert_eq!(table.remove(&1), Some("One"));
    /// assert_eq!(table.contains_key(&1), false);
    /// ```
    /// ```ignore
    /// # use mnemosyne::forktable::ForkTable;
    /// let mut level_1: ForkTable<isize,&str> = ForkTable::new();
    /// level_1.insert(1, "One");
    /// assert_eq!(level_1.contains_key(&1), true);
    ///
    /// let mut level_2: ForkTable<isize,&str> = level_1.fork();
    /// assert_eq!(level_2.chain_contains_key(&1), true);
    /// assert_eq!(level_2.remove(&1), None);
    /// assert_eq!(level_2.chain_contains_key(&1), false);
    /// ```
    pub fn remove(&mut self, key: &K) -> Option<V>
    where K: Clone
    {
        self.whiteouts.insert(key.clone());
        self.table.remove(&key)
    }

    /// Removes a key from this layer's map and whiteouts, so that
    /// definitions of that key from lower levels are exposed.
    ///
    /// Unlike `ForkTable::remove()`, if the removed value exists in a
    /// lower level of the table, it will NOT be whited out. This means
    /// that the definition of that entry from lower levels of the table
    /// will be exposed at this level.
    ///
    /// The key may be any borrowed form of the map's key type, but
    /// `Hash` and `Eq` on the borrowed form *must* match those for
    /// the key type.
```

```
    ///
    /// # Arguments
    ///
    ///  + `key`  - the key to expose
    ///
    /// # Return Value
    ///
    ///  + `Some(V)` if an entry for the given key exists in the
    ///      table, or `None` if there is no entry for that key.
    ///
    pub fn expose<Q: ?Sized>(&mut self, key: &Q) -> Option<V>
    where K: Borrow<Q>
        , Q: Hash + Eq
    {
        self.whiteouts.remove(key);
        self.table.remove(key)
    }

    /// Inserts a key-value pair from the map.
    ///
    /// If the key already had a value present in the map, that
    /// value is returned. Otherwise, `None` is returned.
    ///
    /// If the key is currently whited out (i.e. it was defined
    /// in a lower level of the map and was removed) then it will
    /// be un-whited out and added at this level.
    ///
    /// # Arguments
    ///
    ///  + `k`  - the key to add
    ///  + `v`  - the value to associate with that key
    ///
    /// # Return Value
    ///
    ///  + `Some(V)` if a previous entry for the given key exists in the
    ///      table, or `None` if there is no entry for that key.
    ///
    /// # Examples
    ///
    /// Simply inserting an entry:
    ///
    /// ```ignore
    /// # use mnemosyne::forktable::ForkTable;
    /// let mut table: ForkTable<isize,&str> = ForkTable::new();
    /// assert_eq!(table.get(&1), None);
    /// table.insert(1, "One");
    /// assert_eq!(table.get(&1), Some(&"One"));
    /// ```
    ///
    /// Overwriting the value associated with a key:
    ///
    /// ```ignore
```

```
    /// # use mnemosyne::forktable::ForkTable;
    /// let mut table: ForkTable<isize,&str> = ForkTable::new();
    /// assert_eq!(table.get(&1), None);
    /// assert_eq!(table.insert(1, "one"), None);
    /// assert_eq!(table.get(&1), Some(&"one"));
    ///
    /// assert_eq!(table.insert(1, "One"), Some("one"));
    /// assert_eq!(table.get(&1), Some(&"One"));
    /// ```
    pub fn insert(&mut self, k: K, v: V) -> Option<V> {
        if self.whiteouts.contains(&k) {
            self.whiteouts.remove(&k);
        };
        self.table.insert(k, v)
    }

    /// Returns true if this level contains a value for the specified key.
    ///
    /// The key may be any borrowed form of the map's key type, but
    /// `Hash` and `Eq` on the borrowed form *must* match those for
    /// the key type.
    ///
    /// # Arguments
    ///
    ///  + `k`  - the key to search for
    ///
    /// # Return Value
    ///
    ///  + `true` if the given key is defined in this level of the
    ///    table, `false` if it does not.
    ///
    /// # Examples
    /// ```ignore
    /// # use mnemosyne::forktable::ForkTable;
    /// let mut table: ForkTable<isize,&str> = ForkTable::new();
    /// assert_eq!(table.contains_key(&1), false);
    /// table.insert(1, "One");
    /// assert_eq!(table.contains_key(&1), true);
    /// ```
    /// ```ignore
    /// # use mnemosyne::forktable::ForkTable;
    /// let mut level_1: ForkTable<isize,&str> = ForkTable::new();
    /// assert_eq!(level_1.contains_key(&1), false);
    /// level_1.insert(1, "One");
    /// assert_eq!(level_1.contains_key(&1), true);
    ///
    /// let mut level_2: ForkTable<isize,&str> = level_1.fork();
    /// assert_eq!(level_2.contains_key(&1), false);
    /// ```
    pub fn contains_key<Q: ?Sized>(&self, key: &Q) -> bool
    where K: Borrow<Q>
        , Q: Hash + Eq
```

```
    {
        !self.whiteouts.contains(key) &&
         self.table.contains_key(key)
    }

    /// Returns true if the key is defined in this level of the table, or
    /// in any of its’ parents and is not whited out.
    ///
    /// The key may be any borrowed form of the map’s key type, but
    /// `Hash` and `Eq` on the borrowed form *must* match those for
    /// the key type.
    ///
    /// # Arguments
    ///
    ///  + `k`  - the key to search for
    ///
    /// # Return Value
    ///
    ///  + `true` if the given key is defined in the table,
    ///    `false` if it does not.
    ///
    /// # Examples
    /// ```ignore
    /// # use mnemosyne::forktable::ForkTable;
    /// let mut table: ForkTable<isize,&str> = ForkTable::new();
    /// assert_eq!(table.chain_contains_key(&1), false);
    /// table.insert(1, "One");
    /// assert_eq!(table.chain_contains_key(&1), true);
    /// ```
    /// ```ignore
    /// # use mnemosyne::forktable::ForkTable;
    /// let mut level_1: ForkTable<isize,&str> = ForkTable::new();
    /// assert_eq!(level_1.chain_contains_key(&1), false);
    /// level_1.insert(1, "One");
    /// assert_eq!(level_1.chain_contains_key(&1), true);
    ///
    /// let mut level_2: ForkTable<isize,&str> = level_1.fork();
    /// assert_eq!(level_2.chain_contains_key(&1), true);
    /// ```
    pub fn chain_contains_key<Q:? Sized>(&self, key: &Q) -> bool
    where K: Borrow<Q>
       , Q: Hash + Eq
    {
        self.table.contains_key(key) ||
            (!self.whiteouts.contains(key) &&
                self.parent
                    .map_or(false, |ref p| p.chain_contains_key(key))
                )
    }

    /// Forks this table, returning a new `ForkTable<K,V>`.
    ///
```

```
    /// This level of the table will be set as the child's
    /// parent. The child will be created with an empty backing
    /// `HashMap` and no keys whited out.
    ///
    /// Note that the new `ForkTable<K,V>` has a lifetime
    /// bound ensuring that it will live at least as long as the
    /// parent `ForkTable`.
    pub fn fork(&'a self) -> ForkTable<'a, K, V> {
        ForkTable { table: HashMap::new()
                  , whiteouts: HashSet::new()
                  , parent: Some(self)
                  , level: self.level + 1
                  }
    }

    /// Constructs a new `ForkTable<K,V>`
    pub fn new() -> ForkTable<'a, K,V> {
        ForkTable { table: HashMap::new()
                  , whiteouts: HashSet::new()
                  , parent: None
                  , level: 0
                  }
    }

    /// Wrapper for the backing map's `values()` function.
    ///
    /// Provides an iterator visiting all values in arbitrary
    /// order. Iterator element type is &'b V.
    pub fn values(&self) -> Values<K, V> { self.table.values() }

    /// Wrapper for the backing map's `keys()` function.
    ///
    /// Provides an iterator visiting all keys in arbitrary
    /// order. Iterator element type is &'b K.
    pub fn keys(&self) -> Keys<K, V> { self.table.keys() }
}

/// Allows `table[&key]` indexing syntax.
///
/// This is just a wrapper for `get(&key)`
///
/// ```ignore
/// # use mnemosyne::forktable::ForkTable;
/// let mut table: ForkTable<isize,&str> = ForkTable::new();
/// table.insert(1, "One");
/// assert_eq!(table[&1], "One");
/// ```
impl<'a, 'b, K, Q: ?Sized, V> ops::Index<&'b Q> for ForkTable<'a, K, V>
where K: Borrow<Q>
    , K: Eq + Hash
    , Q: Eq + Hash
{
```

```
    type Output = V;

    #[inline]
    fn index(&self, index: &Q) -> &Self::Output {
        self.get(index)
            .expect_ice("undefined index")
    }

}

/// Allows mutable `table[&key]` indexing syntax.
///
/// This is just a wrapper for `get_mut(&key)`
///
/// ```ignore
/// # use mnemosyne::forktable::ForkTable;
/// let mut table: ForkTable<isize,&str> = ForkTable::new();
/// table.insert(1, "One");
/// table[&1] = "one";
/// assert_eq!(table[&1], "one")
/// ```
impl<'a, 'b, K, Q: ?Sized, V> ops::IndexMut<&'b Q> for ForkTable<'a, K, V>
where K: Borrow<Q>
    , K: Eq + Hash
    , Q: Eq + Hash
{
    #[inline]
    fn index_mut(&mut self, index: &Q) -> &mut V {
        self.get_mut(index)
            .expect_ice("undefined index")
    }

}

#[cfg(test)]
mod tests {
    use super::ForkTable;

    #[test]
    fn test_get_defined() {
        let mut table: ForkTable<isize,&str> = ForkTable::new();
        assert_eq!(table.get(&1), None);
        table.insert(1, "One");
        assert_eq!(table.get(&1), Some(&"One"));
    }

    #[test]
    fn test_get_undefined() {
        let mut table: ForkTable<isize,&str> = ForkTable::new();
        table.insert(1, "One");
        assert_eq!(table.get(&2), None);
    }
```

```
    #[test]
    fn test_get_multilevel() {
        let mut level_1: ForkTable<isize,&str> = ForkTable::new();
        level_1.insert(1, "One");

        let mut level_2: ForkTable<isize,&str> = level_1.fork();
        assert_eq!(level_2.get(&1), Some(&"One"));
    }

    #[test]
    fn test_get_mut_defined() {
        let mut table: ForkTable<isize,&str> = ForkTable::new();
        assert_eq!(table.get_mut(&1), None);
        table.insert(1, "One");
        assert_eq!(table.get_mut(&1), Some(&mut "One"));
    }

    #[test]
    fn test_get_mut_undefined() {
        let mut table: ForkTable<isize,&str> = ForkTable::new();
        table.insert(1, "One");
        assert_eq!(table.get_mut(&2), None);
    }
    #[test]
    fn test_get_mut_multilevel() {
        let mut level_1: ForkTable<isize,&str> = ForkTable::new();
        level_1.insert(1, "One");

        let mut level_2: ForkTable<isize,&str> = level_1.fork();
        assert_eq!(level_2.get_mut(&1), None);
    }
    #[test]
    fn test_remove_returned() {
        let mut table: ForkTable<isize,&str> = ForkTable::new();
        table.insert(1, "One");
        assert_eq!(table.remove(&1), Some("One"));
    }
    #[test]
    fn test_remove_not_defined_after() {
        let mut table: ForkTable<isize,&str> = ForkTable::new();
        table.insert(1, "One");
        table.remove(&1);
        assert_eq!(table.get(&1), None);
    }

    #[test]
    fn test_remove_multilevel() {
        let mut level_1: ForkTable<isize,&str> = ForkTable::new();
        level_1.insert(1, "One");
        assert_eq!(level_1.contains_key(&1), true);

        let mut level_2: ForkTable<isize,&str> = level_1.fork();
```

```
        assert_eq!(level_2.chain_contains_key(&1), true);
        assert_eq!(level_2.remove(&1), None);
        assert_eq!(level_2.chain_contains_key(&1), false);
    }

    #[test]
    fn test_insert_defined_after() {
        let mut table: ForkTable<isize,&str> = ForkTable::new();
        assert_eq!(table.get(&1), None);
        table.insert(1, "One");
        assert_eq!(table.get(&1), Some(&"One"));
    }

    #[test]
    fn test_insert_overwrite() {
        let mut table: ForkTable<isize,&str> = ForkTable::new();
        assert_eq!(table.get(&1), None);
        assert_eq!(table.insert(1, "one"), None);
        assert_eq!(table.get(&1), Some(&"one"));

        assert_eq!(table.insert(1, "One"), Some("one"));
        assert_eq!(table.get(&1), Some(&"One"));
    }

    #[test]
    fn test_contains_key() {
        let mut table: ForkTable<isize,&str> = ForkTable::new();
        assert_eq!(table.contains_key(&1), false);
        table.insert(1, "One");
        assert_eq!(table.contains_key(&1), true);
    }

    #[test]
    fn test_contains_key_this_level_only () {
        let mut level_1: ForkTable<isize,&str> = ForkTable::new();
        assert_eq!(level_1.contains_key(&1), false);
        level_1.insert(1, "One");
        assert_eq!(level_1.contains_key(&1), true);

        let mut level_2: ForkTable<isize,&str> = level_1.fork();
        assert_eq!(level_2.contains_key(&1), false);
    }

    #[test]
    fn test_chain_contains_key_this_level() {
        let mut table: ForkTable<isize,&str> = ForkTable::new();
        assert_eq!(table.chain_contains_key(&1), false);
        table.insert(1, "One");
        assert_eq!(table.chain_contains_key(&1), true);
    }

    #[test]
```

```
    fn test_contains_key_multilevel() {
        let mut level_1: ForkTable<isize,&str> = ForkTable::new();
        assert_eq!(level_1.chain_contains_key(&1), false);
        level_1.insert(1, "One");
        assert_eq!(level_1.chain_contains_key(&1), true);

        let mut level_2: ForkTable<isize,&str> = level_1.fork();
        assert_eq!(level_2.chain_contains_key(&1), true);
    }

    #[test]
    fn test_indexing() {
        let mut table: ForkTable<isize,&str> = ForkTable::new();
        table.insert(1, "One");
        assert_eq!(table[&1], "One");
    }

    #[test]
    fn test_index_mut() {
        let mut table: ForkTable<isize,&str> = ForkTable::new();
        table.insert(1, "One");
        table[&1] = "one";
        assert_eq!(table[&1], "one")
    }
}
```

position.rs

```
//
// Mnemosyne: a functional systems programming language.
// (c) 2015 Hawk Weisman
//
// Mnemosyne is released under the MIT License. Please refer to
// the LICENSE file at the top-level directory of this distribution
// or at https://github.com/hawkw/mnemosyne/.
//

use std::ops::{Deref, DerefMut};
use std::hash;
use std::fmt;
use std::convert::From;

use combine::primitives::SourcePosition;

/// Struct representing a position within a source code file.
///
/// This represents positions using `i32`s because that's how
/// positions are represented in `combine` (the parsing library
/// that we will use for the Mnemosyne parser). I personally would
/// have used `usize`s...
#[derive(Copy, Clone, PartialEq, Eq, Debug, PartialOrd, Ord)]
pub struct Position { pub col: i32
```

````rust
                    , pub row: i32
                    , pub raw: i32
                    }

impl Position {

    /// Create a new `Position `at the given column and row.
    #[inline]
    pub fn new(col: i32, row: i32) -> Self {
        Position { col: col
                 , row: row
                 , raw: col + row
                 }
    }

}

impl From<SourcePosition> for Position {
    /// Create a new `Position` from a `combine` `SourcePosition`.
    ///
    /// # Example
    /// ```ignore
    /// # extern crate combine;
    /// # extern crate mnemosyne;
    /// # use combine::primitives::SourcePosition;
    /// # use mnemosyne::position::Position;
    /// # fn main() {
    /// let sp = SourcePosition { column: 1, line: 1 };
    /// assert_eq!(Position::from(sp), Position::new(1,1));
    /// # }
    /// ```
    fn from(p: SourcePosition) -> Self { Position::new(p.column, p.line) }
}

impl From<(i32,i32)> for Position {
    /// Create a new `Position` from a tuple of i32s.
    ///
    /// # Example
    /// ```ignore
    /// # use mnemosyne::position::Position;
    /// let tuple: (i32,i32) = (1,1);
    /// assert_eq!(Position::from(tuple), Position::new(1,1));
    /// ```
    fn from((col, row): (i32,i32)) -> Self { Position::new(col,row) }
}


impl fmt::Display for Position {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "line {}, column {}", self.row, self.col)
    }
}
````

```rust

/// A pointer to a value with an associated ‘Position‘
#[derive(Clone, Debug)]
pub struct Positional<T> { pub pos: Position
                         , pub value: T
                         }

impl<T> Positional<T> {
    /// Create a new Positional marker at the given position.
    pub fn at(col: i32, row: i32, value: T) -> Positional<T> {
        Positional { pos: Position::new(col, row)
                   , value: value }
    }

    pub fn value(&self) -> &T { &self.value }
}


impl<T> fmt::Display for Positional<T>
where T: fmt::Display {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "{} at {}", self.value, self.pos)
    }
}

/// A positional pointer is still equal to the underlying
/// value even if they have different positions. This is
/// important so that we can test that two identifiers are
/// the same.
impl<T> PartialEq for Positional<T>
where T: PartialEq {
    fn eq(&self, other: &Positional<T>) -> bool {
        self.value == other.value
    }
}

/// If two things are equal, then they better have the same
/// hash as well. Otherwise there will be sadness.
///
/// Homefully this is Ideologically Correct.
impl<T> hash::Hash for Positional<T>
where T: hash::Hash {
    fn hash<H: hash::Hasher>(&self, state: &mut H) {
        self.value.hash(state)
    }
}


/// This is literally just waving my hands for the compiler.
///
/// Hopefully it understands what I mean.
impl<T> Eq for Positional<T>
```

```
where T: Eq
    , T: PartialEq
    {}

impl<T> Deref for Positional<T> {
    type Target = T;
    fn deref(&self) -> &T {
        &self.value
    }
}

impl<T> DerefMut for Positional<T> {
    fn deref_mut(&mut self) -> &mut T {
        &mut self.value
    }
}

#[cfg(test)]
mod tests {
    use super::*;
    use combine::primitives::SourcePosition;

    #[test]
    fn test_from_sourceposition() {
        let sp = SourcePosition { column: 1, line: 1 };
        assert_eq!(Position::from(sp), Position::new(1,1));
    }

    #[test]
    fn test_from_tuple() {
        let tuple: (i32,i32) = (1,1);
        assert_eq!(Position::from(tuple), Position::new(1,1));
    }
}
```

compile/mod.rs

```
//
// Mnemosyne: a functional systems programming language.
// (c) 2015 Hawk Weisman
//
// Mnemosyne is released under the MIT License. Please refer to
// the LICENSE file at the top-level directory of this distribution
// or at https://github.com/hawkw/mnemosyne/.
//

use std::ffi::CString;
use std::cmp::Ordering;
use std::mem;

```

```
use libc::c_uint;
use llvm_sys::prelude::LLVMValueRef;

use iron_llvm::core;
use iron_llvm::core::types::{ Type
                            , TypeCtor
                            , RealTypeCtor
                            , RealTypeRef
                            , IntTypeCtor
                            , IntTypeRef
                            };
use iron_llvm::{LLVMRef, LLVMRefCtor};

use errors::ExpectICE;
use forktable::ForkTable;
use position::Positional;
use ast::{ Node
         , Form
         , DefForm
         , Ident
         , Function };

use semantic::annotations::{ ScopedState
                           , Scoped
                           };
use semantic::types;
use semantic::types::{ Primitive
                     , Reference
                     };

/// Result type for compiling an AST node to LLVM IR
///
/// An `IRResult` contains either a `ValueRef`, if compilation was successful,
/// or a `Positional<String>` containing an error message and the position of
/// the line of code which could not be compiled.
pub type IRResult = Result<LLVMValueRef, Vec<Positional<String>>>;

/// Result type for compiling a type to an LLVM `TypeRef`.
pub type TypeResult<T: Type + Sized> = Result<T, Positional<String>>;

pub type NamedValues<'a> = ForkTable<'a, &'a str, LLVMValueRef>;

#[inline] fn word_size() -> usize { mem::size_of::<isize>() }

/// Trait for that which may join in The Great Work
pub trait Compile {
    /// Compile `self` to an LLVM `ValueRef`
    ///
    /// # Returns:
    ///   - `Ok` containing a `ValueRef` if this was compiled correctly.
    ///   - An `Err` with a vector of error messages containing any
    ///     errors that occured during compilation.
```

```
    ///
    /// # Panics:
    ///   - If something has gone horribly wrong. This does NOT panic if the
    ///     code could not be compiled because it was incorrect, but it will
    ///     panic in the event of an internal compiler error.
    fn to_ir(&self, context: LLVMContext) -> IRResult;
}

/// Trait for type tags that can be translated to LLVM
// pub trait TranslateType {
//     /// Translate `self` to an LLVM `TypeRef`
//     ///
//     /// # Returns:
//     ///   - `Ok` containing a `TypeRef` if this was compiled correctly.
//     ///   - An `Err` with a positional error message in the event of
//     ///     a type error.
//     ///
//     /// # Panics:
//     ///   - In the event of an internal compiler error (i.e. if a well-formed
//     ///     type could not be gotten from LLVM correctly).
//     fn translate_type(&self, context: LLVMContext) -> TypeResult;
// }

/// LLVM compilation context.
///
/// This is based rather loosely on MIT License code from
/// the [iron-kaleidoscope](https://github.com/jauhien/iron-kaleidoscope)
/// tutorial, and from [`librustc_trans`](https://github.com/rust-lang/rust/blob/master/src/libr
/// from the Rust compiler.
pub struct LLVMContext<'a> { llctx: core::Context
                           , llmod: core::Module
                           , llbuilder: core::Builder
                           , named_vals: NamedValues<'a>
                           }

/// because we are in the Raw Pointer Sadness Zone (read: unsafe),
/// it is necessary that we assert that everything exists.
macro_rules! not_null {
    ($target:expr) => ({
        let e = $target;
        if e.is_null() {
            ice!( "assertion failed: {} returned null!"
                , stringify!($target)
                );
        } else { e }
    })
}

/// converts a raw pointer that may be null to an Option
/// the compiler will yell about this, claiming that it involves
/// an unused unsafe block, but the unsafe block is usually necessary.
macro_rules! optionalise {
```

```
    ($target:expr) => ({
            let e = unsafe { $target };
            if e.is_null() {
                None
            } else { Some(e) }
    })
}

macro_rules! try_vec {
    ($expr:expr) => ({
        if !$expr.is_empty() {
            return Err($expr)
        }
    })
}

// ------------ SEGFAULT EXISTS SOMEWHERE BELOW THIS LINE --------------------

//
// impl<'a> LLVMContext<'a> {
//
//     /// Constructs a new LLVM context.
//     ///
//     /// # Returns:
//     ///   - An `LLVMContext`
//     ///
//     /// # Panics:
//     ///   - If the LLVM C ABI returned a null value for the `Context`,
//     ///     `Builder`, or `Module`
//     pub fn new(module_name: &str) -> Self {
//         LLVMContext { llctx: core::Context::get_global()
//                     , llmod: core::Module::new(module_name)
//                     , llbuilder: core::Builder::new()
//                     , named_vals: NamedValues::new()
//                     }
//     }
//
//     /// Dump the module's contents to stderr for debugging
//     ///
//     /// Apparently this is the only reasonable way to get a textual
//     /// representation of a `Module` in LLVM
//     pub fn dump(&self) { self.llmod.dump() }
//
//     pub fn int_type(&self, size: usize) -> IntTypeRef {
//         IntTypeRef::get_int_in_context(&self.llctx, size as c_uint)
//     }
//
//     pub fn float_type(&self) -> RealTypeRef {
//         RealTypeRef::get_float_in_context(&self.llctx)
//     }
//     pub fn double_type(&self) -> RealTypeRef {
//         RealTypeRef::get_double_in_context(&self.llctx)
```

```
//     }
//     pub fn byte_type(&self) -> IntTypeRef {
//         IntTypeRef::get_int8_in_context(&self.llctx)
//     }
//
//     /// Get any existing declarations for a given function name.
//     ///
//     /// # Returns:
//     ///   - ‘Some‘ if there is an existing previous declaration
//     ///     for this function.
//     ///   - ‘None‘ if the function has not been declared previously.
//     ///
//     /// # Panics:
//     ///   - If the C string representation for the function name could
//     ///     not be created.
//     pub fn get_fn(&self, name: &Ident) -> Option<core::FunctionRef> {
//         self.llmod.get_function_by_name(name.value.as_ref())
//     }
// }

// impl<’a> Compile for Scoped<’a, Form<’a, ScopedState>> {
//     fn to_ir(&self, context: LLVMContext) -> IRResult {
//         match **self {
//             Form::Define(ref form) => unimplemented!()
//           , Form::Let(ref form) => unimplemented!()
//           , Form::If { .. } => unimplemented!()
//           , Form::Call { .. } => unimplemented!()
//           , Form::Lambda(ref fun) => unimplemented!()
//           , Form::Logical(ref exp) => unimplemented!()
//           , Form::Lit(ref c) => unimplemented!()
//           , Form::NameRef(ref form) => unimplemented!()
//         }
//     }
// }
//
// impl<’a> Compile for Scoped<’a, DefForm<’a, ScopedState>> {
//     fn to_ir(&self, context: LLVMContext) -> IRResult {
//         match **self {
//             DefForm::TopLevel { ref name, ref value, .. } =>
//                 unimplemented!()
//          ,  DefForm::Function { ref name, ref fun } => {
//                 match context.get_fn(name) {
//                     Some(previous) => unimplemented!()
//                   , None => unimplemented!()
//                 }
//             }
//         }
//     }
// }
//
//
// impl<’a> Compile for Scoped<’a, Function<’a, ScopedState>> {
```

```
//
//     fn to_ir(&self, context: LLVMContext) -> IRResult {
//         let mut errs: Vec<Positional<String>> = vec![];
//         // Check to see if the pattern binds an equivalent number of arguments
//         // as the function signature (minus one, which is the return type).
//         for e in &self.equations {
//             match e.pattern_length()
//                    .cmp(&self.arity()) {
//                 // the equation's pattern is shorter than the function's arity
//                 // eventually, we'll autocurry this, but for now, we error.
//                 // TODO: maybe there should be a warning as well?
//                 Ordering::Less => errs.push(Positional {
//                     pos: e.position.clone()
//                   , value: format!( "[error] equation had fewer bindings \
//                                      than function arity\n \
//                                      [error] auto-currying is not currently \
//                                      implemented.\n \
//                                      signature: {}\nfunction: {}\n"
//                                   , self.sig
//                                   , (*e).to_sexpr(0)
//                                   )
//                   })
//                 // the equation's pattern is longer than the function's arity
//                 // this is super wrong and always an error.
//               , Ordering::Greater => errs.push(Positional {
//                   pos: e.position.clone()
//                 , value: format!( "[error] equation bound too many arguments\n \
//                                    signature: {}\nfunction: {}\n"
//                                 , self.sig
//                                 , (*e).to_sexpr(0)
//                                 )
//               })
//             , _ =>  {}
//             }
//         }
//         // TODO: this could be made way more idiomatic...
//         try_vec!(errs);
//         unimplemented!()
//     }
//
// }
//
//
//
// // impl TranslateType for types::Type {
// //     fn translate_type(&self, context: LLVMContext) -> TypeResult {
// //         match *self {
// //             types::Type::Ref(ref r) => r.translate_type(context)
// //           , types::Type::Prim(ref p) => p.translate_type(context)
// //           , _ => unimplemented!() // TODO: figure this out
// //         }
// //     }
```

```
// // }
//
// // impl TranslateType for Reference {
// //     fn translate_type(&self, context: LLVMContext) -> TypeResult {
// //         unimplemented!() // TODO: figure this out
// //     }
// // }
//
// // impl TranslateType for Primitive {
// //     fn translate_type(&self, context: LLVMContext) -> TypeResult {
// //     //     Ok(match *self {
// //     //         Primitive::IntSize => context.int_type(word_size())
// //     //       , Primitive::UintSize => context.int_type(word_size())
// //     //       , Primitive::Int(bits) => context.int_type(bits as usize)
// //     //       , Primitive::Uint(bits) => context.int_type(bits as usize)
// //     //       , Primitive::Float => context.float_type()
// //     //       , Primitive::Double => context.double_type()
// //     //       , Primitive::Byte => context.byte_type()
// //     //       , _ => unimplemented!() // TODO: figure this out
// //     //   })
// //         unimplemented!()
// //     }
// // }
```

B.2 Mnemosyne Parser Crate

lib.rs

```
//
// Mnemosyne: a functional systems programming language.
// (c) 2015 Hawk Weisman
//
// Mnemosyne is released under the MIT License. Please refer to
// the LICENSE file at the top-level directory of this distribution
// or at https://github.com/hawkw/mnemosyne/.
//

extern crate combine;
extern crate combine_language;
extern crate mnemosyne as core;

use combine::*;
use combine_language::{ LanguageEnv
                      , LanguageDef
                      , Identifier
                      };
use combine::primitives::{ Stream
                         , Positioner
                         , SourcePosition
                         };
use core::chars;
```

```
use core::semantic::*;
use core::semantic::annotations::{ Annotated
                                 , UnscopedState
                                 , Unscoped
                                 };
use core::semantic::types::*;
use core::semantic::ast::*;
use core::position::*;

use std::rc::Rc;

type ParseFn<'a, I, T> = fn (&MnEnv<'a, I>, State<I>) -> ParseResult<T, I>;

type U = UnscopedState;

mod tests;

/// Wraps a parsing function with a language definition environment.
///
/// TODO: this could probably push identifiers to the symbol table here?
#[derive(Copy)]
struct MnParser<'a: 'b, 'b, I, T>
where I: Stream<Item=char>
    , I::Range: 'b
    , I: 'b
    , I: 'a
    , T: 'a {
        env: &'b MnEnv<'a, I>
      , parser: ParseFn<'a, I, T>
}

impl<'a, 'b, I, T> Clone for MnParser<'a, 'b, I, T>
where I: Stream<Item=char>
    , I::Range: 'b
    , I: 'b
    , T: 'a
    , 'a: 'b {

    fn clone(&self) -> Self {
        MnParser { env: self.env , parser: self.parser }
    }
}

impl<'a, 'b, I, T> Parser for MnParser<'a, 'b, I, T>
where I: Stream<Item=char>
    , I::Range: 'b
    , I: 'b
    , T: 'a
    , 'a: 'b {

    type Input = I;
    type Output = T;
```

```

    fn parse_state(&mut self, input: State<I>) -> ParseResult<T, I> {
        (self.parser)(self.env, input)
    }

}

struct MnEnv<'a, I>
where I: Stream<Item = char>
    , I::Item: Positioner<Position = SourcePosition>
    , I: 'a {
    env: LanguageEnv<'a, I>
}

impl <'a, I> std::ops::Deref for MnEnv<'a, I>
where I: Stream<Item=char>
    , I: 'a {
    type Target = LanguageEnv<'a, I>;
    fn deref(&self) -> &LanguageEnv<'a, I> { &self.env }
}

impl<'a, 'b, I> MnEnv<'a, I>
where I: Stream<Item=char>
    , I::Item: Positioner<Position = SourcePosition>
    , I::Range: 'b {

    /// Wrap a function into a MnParser with this environment
    fn parser<T>(&'b self, parser: ParseFn<'a, I, T>)
                -> MnParser<'a, 'b, I, T> {
        MnParser { env: self, parser: parser }
    }

    #[allow(dead_code)]
    fn parse_def(&self, input: State<I>) -> ParseResult<Form<'a, U>, I> {
        let function_form
            = self.name()
                  .and(self.function())
                  .map(|(name, fun)| DefForm::Function { name: name
                                                       , fun: fun });
        let top_level
            = self.name()
                  .and(self.type_name())
                  .and(self.expr())
                  .map(|((name, ty), body)|
                    DefForm::TopLevel { name: name
                                      , annot: ty
                                      , value: Rc::new(body) });

        self.reserved("def").or(self.reserved("define"))
            .with(function_form.or(top_level))
            .map(Form::Define)
            .parse_state(input)
```

```
    }

    #[allow(dead_code)]
    fn parse_if(&self, input: State<I>) -> ParseResult<Form<'a, U>, I> {
        self.reserved("if")
            .with(self.expr())
            .and(self.expr())
            .and(optional(self.expr()))
            .map(|((cond, if_clause), else_clause)|
                Form::If { condition: Rc::new(cond)
                         , if_clause: Rc::new(if_clause)
                         , else_clause: else_clause.map(Rc::new)
                         })
            .parse_state(input)
    }

    #[allow(dead_code)]
    fn parse_lambda(&self, input: State<I>) -> ParseResult<Form<'a, U>, I> {
        self.reserved("lambda")
            .or(self.reserved(chars::LAMBDA))
            .with(self.function())
            .map(Form::Lambda)
            .parse_state(input)
    }

    #[allow(dead_code)]
    fn parse_function(&self, input: State<I>) -> ParseResult<Function<'a, U>, I> {
        let fn_kwd = choice([ self.reserved("fn")
                            , self.reserved("lambda")
                            , self.reserved(chars::LAMBDA)
                            ]);

        self.parens(fn_kwd.with(
            self.signature()
                .and(many1(self.equation()))
                .map(|(sig, eqs)| Function { sig: sig
                                         , equations: eqs
                                         })
                ))
            .parse_state(input)
    }

    #[allow(dead_code)]
    fn parse_primitive_ty(&self, input: State<I>) -> ParseResult<Type, I> {
        choice([ self.reserved("int")
                     .with(value(Primitive::IntSize))
               , self.reserved("uint")
                     .with(value(Primitive::IntSize))
               , self.reserved("float")
                     .with(value(Primitive::Float))
               , self.reserved("double")
                     .with(value(Primitive::Double))
```

```
               , self.reserved("bool")
                     .with(value(Primitive::Bool))
               , self.reserved("i8")
                     .with(value(Primitive::Int(Int::Int8)))
               , self.reserved("i16")
                     .with(value(Primitive::Int(Int::Int16)))
               , self.reserved("i32")
                    .with(value(Primitive::Int(Int::Int32)))
               , self.reserved("i64")
                     .with(value(Primitive::Int(Int::Int64)))
              , self.reserved("u8")
                    .with(value(Primitive::Uint(Int::Int8)))
              , self.reserved("u16")
                    .with(value(Primitive::Uint(Int::Int16)))
              , self.reserved("u32")
                    .with(value(Primitive::Uint(Int::Int32)))
              , self.reserved("u64")
                    .with(value(Primitive::Uint(Int::Int64)))
               ])
               .map(|primitive| Type::Prim(primitive))
               .parse_state(input)
    }

    pub fn raw_ptr_ty(&self, input: State<I>) -> ParseResult<Type, I> {
        char('*').with(self.type_name())
                 .map(|t| Type::Ref(Reference::Raw(Rc::new(t))))
                 .parse_state(input)
    }

    pub fn unique_ptr_ty(&self, input: State<I>) -> ParseResult<Type, I> {
        char('@').with(self.type_name())
                 .map(|t| Type::Ref(Reference::Unique(Rc::new(t))))
                 .parse_state(input)
    }

    pub fn borrow_ptr_ty(&self, input: State<I>) -> ParseResult<Type, I> {
        char('&').with(self.type_name())
                 .map(|t| Type::Ref(Reference::Borrowed(Rc::new(t))))
                 .parse_state(input)
    }
    fn parse_type(&self, input: State<I>) -> ParseResult<Type, I> {
        choice([ self.parser(MnEnv::parse_primitive_ty)
               , self.parser(MnEnv::raw_ptr_ty)
               , self.parser(MnEnv::unique_ptr_ty)
               , self.parser(MnEnv::borrow_ptr_ty)
               ])
            .parse_state(input)
    }

    fn parse_name_deref(&self, input: State<I>) -> ParseResult<NameRef, I> {
        char('*').with(self.name())
                 .map(NameRef::Deref)
```

```
                .parse_state(input)
    }

    fn parse_name_unique(&self, input: State<I>) -> ParseResult<NameRef, I> {
        char('@').with(self.name())
                .map(NameRef::Unique)
                .parse_state(input)
    }

    fn parse_name_borrow(&self, input: State<I>)-> ParseResult<NameRef, I> {
        char('&').with(self.name())
                .map(NameRef::Borrowed)
                .parse_state(input)
    }

    fn parse_owned_name(&self, input: State<I>) -> ParseResult<NameRef, I> {
        self.name()
            .map(NameRef::Owned)
            .parse_state(input)
    }

    fn parse_name_ref(&self, input: State<I>)
                    -> ParseResult<Form<'a, U>, I> {
        choice([ self.parser(MnEnv::parse_name_deref)
               , self.parser(MnEnv::parse_name_unique)
               , self.parser(MnEnv::parse_name_borrow)
               , self.parser(MnEnv::parse_owned_name)
               ])
            .map(Form::NameRef)
            .parse_state(input)
    }

    // fn parse_typeclass_arrow(&self, input: State<I>) -> ParseResult<&str, I> {
    //     self.reserved_op("=>")
    //         .or(self.reserved_op(FAT_ARROW))
    //         .parse_state(input)
    // }

    // fn parse_arrow(&self, input: State<I>) -> ParseResult<&str, I> {
    //     self.reserved_op("->")
    //         .or(self.reserved_op(ARROW))
    //         .parse_state(input)
    // }

    fn parse_prefix_constraint(&self, input: State<I>)
                             -> ParseResult<Constraint, I> {
        self.parens(self.reserved_op("=>")
                        .or(self.reserved_op(chars::FAT_ARROW))
                        .with(self.name())
                        .and(many1(self.name())) )
            .map(|(c, gs)| Constraint { typeclass: c
                                      , generics: gs })
```

```
            .parse_state(input)
    }

    fn parse_infix_constraint(&self, input: State<I>)
                              -> ParseResult<Constraint, I> {
        self.braces(self.name()
                        .skip(self.reserved_op("=>")
                                  .or(self.reserved_op(chars::FAT_ARROW)))
                        .and(many1(self.name())) )
            .map(|(c, gs)| Constraint { typeclass: c
                                      , generics: gs })
            .parse_state(input)
    }

    fn parse_constraint(&self, input: State<I>)
                              -> ParseResult<Constraint, I> {
        self.parser(MnEnv::parse_prefix_constraint)
            .or(self.parser(MnEnv::parse_infix_constraint))
            .parse_state(input)
    }

    pub fn constraint(&'b self) -> MnParser<'a, 'b, I, Constraint> {
        self.parser(MnEnv::parse_constraint)
    }

    fn parse_prefix_sig(&self, input: State<I>) -> ParseResult<Signature, I> {
        self.parens(self.reserved_op("->")
                        .or(self.reserved_op(chars::ARROW))
                        .with(optional(many1(self.constraint())))
                        .and(many1(self.type_name())) )
            .map(|(cs, glob)| Signature { constraints: cs
                                        , typechain: glob })
            .parse_state(input)
    }

    fn parse_infix_sig(&self, input: State<I>) -> ParseResult<Signature, I> {
        self.braces(optional(many1(self.constraint()))
                        .and(sep_by1::< Vec<Type>
                                      , _, _>( self.lex(self.type_name())
                                             , self.reserved_op("->")
                                                   .or(self.reserved_op(
                                                       chars::ARROW)
                                                   )
                                             )))
            .map(|(cs, glob)| Signature { constraints: cs
                                        , typechain: glob })
            .parse_state(input)
    }

    fn parse_signature(&self, input: State<I>) -> ParseResult<Signature, I> {

        // let prefix =
```

```
        //     self.parens(self.reserved_op("->")
        //                     .or(self.reserved_op(ARROW))
        //                     .with(optional(many1(self.constraint())))
        //                     .and(many1(self.type_name())) )
        //         .map(|(cs, glob)| Signature { constraints: cs
        //                                     , typechain: glob });
        //
        // let infix =
        //     self.braces(optional(many1(self.constraint()))
        //                     .and(sep_by1::< Vec<Type>
        //                                  , _, _>( self.lex(self.type_name())
        //                                         , self.reserved_op("->")
        //                                               .or(self.reserved_op(ARROW))
        //                                         )))
        //         .map(|(cs, glob)| Signature { constraints: cs
        //                                     , typechain: glob });
        // prefix.or(infix)
        //       .parse_state(input)
        self.parser(MnEnv::parse_prefix_sig)
            .or(self.parser(MnEnv::parse_infix_sig))
            .parse_state(input)
    }

    pub fn signature(&'b self) -> MnParser<'a, 'b, I, Signature> {
        self.parser(MnEnv::parse_signature)
    }

    fn parse_binding(&self, input: State<I>)
                    -> ParseResult<Unscoped<'a, Binding<'a, U>>, I> {
        let pos = input.position.clone();
        self.parser(MnEnv::parse_name)
            .and(self.type_name())
            .and(self.expr())
            .map(|((name, typ), value)|
                Annotated::new( Binding { name: name
                                        , typ: typ
                                        , value: Rc::new(value)
                                        }
                              , Position::from(pos)
                          ))
            .parse_state(input)
    }

    #[allow(dead_code)]
    fn parse_logical(&self, input: State<I>)
                    -> ParseResult<Logical<'a, U>, I> {
        let and = self.reserved("and")
                      .with(self.expr())
                      .and(self.expr())
                      .map(|(a, b)| Logical::And { a: Rc::new(a)
                                                 , b: Rc::new(b)
                                                 });
```

```
         let or = self.reserved("or")
                      .with(self.expr())
                      .and(self.expr())
                      .map(|(a, b)| Logical::And { a: Rc::new(a)
                                                 , b: Rc::new(b)
                                                 });

        and.or(or)
           .parse_state(input)
    }

    pub fn int_const(&'b self) -> MnParser<'a, 'b, I, Literal> {
        self.parser(MnEnv::parse_int_const)
    }

    #[allow(dead_code)]
    fn parse_int_const(&self, input: State<I>) -> ParseResult<Literal, I> {
        self.integer()
            .map(Literal::IntConst)
            .parse_state(input)
    }

    fn parse_let(&self, input: State<I>) -> ParseResult<Form<'a, U>, I> {

        let binding_form =
            self.reserved("let")
                .with(self.parens(many(self.parens(self.binding()))))
                .and(many(self.expr()))
                .map(|(bindings, body)| LetForm::Let { bindings: bindings
                                                      , body: body });

        choice([ binding_form ])
            .map(Form::Let)
            .parse_state(input)
    }

    fn parse_name (&self, input: State<I>) -> ParseResult<Ident, I> {
        let position = input.position.clone();
        self.env.identifier::<'b>()
            .map(|name| Positional { pos: Position::from(position)
                                   , value: name })
            .parse_state(input)
    }

    fn parse_call(&self, input: State<I>) -> ParseResult<Form<'a, U>, I> {
        self.name()
            .and(many(self.expr()))
            .map(|(name, args)| Form::Call { fun: name, body: args })
            .parse_state(input)
    }
```

```
fn parse_expr(&self, input: State<I>) -> ParseResult<Expr<'a, U>, I> {
    let pos = Position::from(input.position.clone());
    self.env.parens(choice([ try(self.call())
                           , try(self.def())
                           , try(self.if_form())
                           , try(self.lambda())
                           , try(self.let_form())
                           ]))
        .or(try(self.int_const()
                    .map(Form::Lit)))
        .or(try(self.name_ref()))
        .map(|f| Annotated::new(f, pos) )
        .parse_state(input)
}

fn parse_pattern(&self, input: State<I>) -> ParseResult<Pattern, I> {
    let pat_elem =
        self.name().map(PatElement::Name)
            .or(self.int_const().map(PatElement::Lit));

    self.parens(many(pat_elem))
        .parse_state(input)
}

pub fn pattern(&'b self) -> MnParser<'a, 'b, I, Pattern> {
    self.parser(MnEnv::parse_pattern)
}

fn parse_equation(&self, input: State<I>)
                 -> ParseResult< Annotated< 'a
                                          , Equation< 'a, U>
                                          , U>
                               , I> {
    let pos = Position::from(input.position.clone());
    self.parens(self.pattern()
                    .and(many(self.expr())))
        .map(|(pat, body)| Annotated::new( Equation { pattern: pat
                                                     , body: body }
                                          , pos ))
        .parse_state(input)
}

pub fn equation(&'b self) -> MnParser< 'a, 'b, I
                                     , Annotated< 'a
                                                , Equation<'a, U>
                                                , U>
                                     > {
    self.parser(MnEnv::parse_equation)
}

pub fn expr(&'b self) -> MnParser<'a, 'b, I, Expr<'a, U>> {
    self.parser(MnEnv::parse_expr)
```

```
    }

    pub fn def(&'b self) -> MnParser<'a, 'b, I, Form<'a, U>> {
        self.parser(MnEnv::parse_def)
    }

    pub fn if_form(&'b self) -> MnParser<'a, 'b, I, Form<'a, U>> {
        self.parser(MnEnv::parse_if)
    }

    pub fn let_form(&'b self) -> MnParser<'a, 'b, I, Form<'a, U>> {
        self.parser(MnEnv::parse_let)
    }

    pub fn lambda(&'b self)-> MnParser<'a, 'b, I, Form<'a, U>> {
        self.parser(MnEnv::parse_lambda)
    }

    pub fn call(&'b self) -> MnParser<'a, 'b, I, Form<'a, U>> {
        self.parser(MnEnv::parse_call)
    }

    pub fn name(&'b self) -> MnParser<'a, 'b, I, Ident> {
        self.parser(MnEnv::parse_name)
    }

    pub fn name_ref(&'b self) -> MnParser<'a, 'b, I, Form<'a, U>> {
        self.parser(MnEnv::parse_name_ref)
    }


    pub fn binding(&'b self)
            -> MnParser< 'a, 'b, I, Unscoped<'a, Binding<'a, U>>> {
        self.parser(MnEnv::parse_binding)
    }

    pub fn type_name(&'b self) -> MnParser<'a, 'b, I, types::Type> {
        self.parser(MnEnv::parse_type)
    }

    pub fn function(&'b self) -> MnParser<'a, 'b, I, Function<'a, U>> {
        self.parser(MnEnv::parse_function)
    }

}
pub fn parse_module<'a>(code: &'a str)
                        -> Result< Vec<Expr<'a, UnscopedState>>
                                 , ParseError<&'a str>>
 {
    let env = LanguageEnv::new(LanguageDef {
        ident: Identifier {
            start: letter().or(satisfy(move |c| chars::ALPHA_EXT.contains(c)))
```

```
          , rest: alpha_num().or(satisfy(move |c| chars::ALPHA_EXT.contains(c)))
          , reserved: [ // a number of these reserved words have no meaning yet
                        "and"                 , "begin"
                      , "case"                , "cond"        , "class"
                      , "data"
                      , "define"              , "defn"        , "def"
                      , "delay"               , "fn"
                      , "do"                  , "else"
                      , "if"                  , "lambda"      , chars::LAMBDA
                      , "let"                 , "let*"        , "letrec"
                      , "or"
                      , "quasiquote"          , "quote"       , "unquote"
                      , "set!"                , "unquote-splicing"
                      , "struct"              , "union"
                      , "i8"                  , "u8"
                      , "i16"                 , "u16"
                      , "i32"                 , "u32"         , "f32"
                      , "i64"                 , "u64"         , "f64"
                      , "int"                 , "uint"        , "float"
                      , "bool"                                , "double"
                      , "ref"                 , "move"        , "borrow"
                      , "trait"               , "typeclass"
                      , "instance"            , "impl"
                      ].iter().map(|x| (*x).into())
                       .collect()
        }
      , op: Identifier {
            start: satisfy(move |c| chars::OPS.contains(c))
          , rest:  satisfy(move |c| chars::OPS.contains(c))
          , reserved: [ "=>", "->", "\\", "|", chars::ARROW, chars::FAT_ARROW]
                .iter().map(|x| (*x).into()).collect()
        }
      , comment_line: string(";").map(|_| ())
      , comment_start: string("#|").map(|_| ())
      , comment_end: string("|#").map(|_| ())
    });
    let env = MnEnv { env: env };

    env.white_space()
       .with(many1::<Vec<Expr<'a, U>>, _>(env.expr()))
       .parse(code)
       .map(|(e, _)| e)
}
```

tests.rs

```
use super::parse_module;

use core::semantic::ast::Node;

macro_rules! expr_test {
    ($name:ident, $code:expr) => {
```

```
        #[test]
        fn $name() {
            assert_eq!( parse_module($code)
                            .unwrap()[0]
                            .to_sexpr(0)
                      , $code)
        }
    }
}

expr_test!(test_basic_add, "(+ 1 2)");
expr_test!(test_basic_sub, "(- 3 4)");
expr_test!(test_basic_div, "(/ 5 6)");
expr_test!(test_basic_mul, "(* 1 2)");
expr_test!(test_nested_arith_1, "(+ 1 (- 2 3))");
expr_test!(test_nested_arith_2, "(* (+ 1 2) 3 4)");
expr_test!(test_nested_arith_3, "(+ (/ 1 2) (* 3 4))");

expr_test!(test_call_1, "(my_fn 1 2)");
expr_test!(test_call_2, "(my_fn (my_other_fn a_var a_different_var))");
expr_test!(test_call_3
  , "(my_fn (my_other_fn a_var a_different_var) VarWithUppercase Othervar)");
expr_test!(test_call_4
  , "(my_fn (my_other_fn a_var a_different_var) (another_fn a_var))");

expr_test!(test_call_ptr_1, "(my_fn a *b)");
expr_test!(test_call_ptr_2, "(my_fn *a *b)");
expr_test!(test_call_ptr_3, "(my_fn &a)");
expr_test!(test_call_ptr_4, "(my_fn a &b)");
expr_test!(test_call_ptr_5, "(my_fn &a &b)");
expr_test!(test_call_ptr_6, "(my_fn @a)");
expr_test!(test_call_ptr_7, "(my_fn a @b)");
expr_test!(test_call_ptr_8, "(my_fn @a @b)");

expr_test!(test_defsyntax_1,
"(define fac (\u{3bb} (\u{8594} int int)
\t((0) 1)
\t((n) (fac (- n 1))))\n)");

#[test]
fn test_defsyntax_sugar() {
    let string =
r#"(def fac (fn {int -> int}
    ((0) 1)
    ((n) (fac (- n 1)))))"#;
    assert_eq!( parse_module(string).unwrap()[0]
                                    .to_sexpr(0)
              , "(define fac (\u{3bb} (\u{8594} int int)
\t((0) 1)
\t((n) (fac (- n 1))))\n)" )
}
```

B.3 Manganese Application Crate

main.rs

```rust
//
// The Manganese Mnemosyne Compilation System
// (c) 2015 Hawk Weisman
//
// Mnemosyne is released under the MIT License. Please refer to
// the LICENSE file at the top-level directory of this distribution
// or at https://github.com/hawkw/mnemosyne/.
//
extern crate clap;
extern crate mnemosyne;
extern crate mnemosyne_parser as parser;

use clap::{Arg, App, SubCommand};

use std::error::Error;
use std::io::Read;
use std::fs::File;
use std::path::PathBuf;

use mnemosyne::ast;
use mnemosyne::ast::Node;
use mnemosyne::errors::UnwrapICE;

const VERSION_MAJOR: u32 = 0;
const VERSION_MINOR: u32 = 1;

fn main() {
    let matches = App::new("Manganese")
        .version(&format!("v{}.{} for {} ({})"
                , VERSION_MAJOR
                , VERSION_MINOR
                , mnemosyne::mnemosyne_version()
                , mnemosyne::llvm_version()
            ))
        .author("Hawk Weisman <hi@hawkweisman.me>")
        .about("[Mn] Manganese: The Mnemosyne Compilation System")
        .args_from_usage(
            "<INPUT> 'Source code file to compile'
             -d, --debug 'Display debugging information'")
        .get_matches();

    let path = matches.value_of("INPUT")
                      .map(PathBuf::from)
                      .unwrap();

    let code = File::open(&path)
        .map_err(|error    | String::from(error.description()) )
        .and_then(|mut file| {
                let mut s = String::new();
```

```
                  file.read_to_string(&mut s)
                      .map_err(|error| String::from(error.description()) )
                      .map(|_| s)
              })
          .unwrap();

     let ast = parser::parse_module(code.as_ref())
                      .unwrap();

    for node in ast { println!("{}", (*node).to_sexpr(0)) }
}
```

Bibliography

If I have seen further, it is by standing on the shoulders of giants.

Letter to Robert Hooke, 1676
SIR ISAAC NEWTON

[1] Jonathan Aldrich et al. "Typestate-oriented Programming". In: *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. OOPSLA '09. Orlando, Florida, USA: ACM, 2009, pp. 1015–1022. ISBN: 978-1-60558-768-4. DOI: 10.1145/1639950.1640073. URL: http://doi.acm.org/10.1145/1639950.1640073.

[2] 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 p. 11).

[3] 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 p. 11).

[4] 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. 2).

[5] 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. 2, 7).

[6] Jim Blandy. *Why Rust?* 1st ed. 1005 Gravenstein Highway North, Sebastopol, CA 95472.: O'Reilly Media, Inc, Sept. 2015. ISBN: 978-1-491-92730-4 (cit. on pp. 7, 12).

[7] Eugene Burmako. "Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming". In: *Proceedings of the 4th Workshop on Scala*. SCALA '13. Montpellier, France: ACM, 2013, 3:1–3:10. ISBN: 978-1-4503-2064-1. DOI: 10.1145/2489837.2489840. URL: http://doi.acm.org/10.1145/2489837.2489840.

[8] 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. 12).

[9] Jeremy Condit et al. "Dependent types for low-level programming". In: *Programming Languages and Systems*. Springer, 2007, pp. 520–535.

[10] 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. 11).

[11] 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.

[12] Matthew Davis et al. "Towards Region-based Memory Management for Go". In: *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. MSPC '12. Beijing, China: ACM, 2012, pp. 58–67. ISBN: 978-1-4503-1219-6. DOI: 10.1145/2247684.2247695. URL: http://doi.acm.org/10.1145/2247684.2247695.

[13] 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. 2).

[14] 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. 12).

[15] 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. 12).

[16] Jeroen Fokker. "Functional parsers". In: *Advanced Functional Programming*. Springer, 1995, pp. 1–23.

[17] Bryan Ford. "Packrat Parsing:: Simple, Powerful, Lazy, Linear Time, Functional Pearl". In: *SIGPLAN Not.* 37.9 (Sept. 2002), pp. 36–47. ISSN: 0362-1340. DOI: 10.1145/583852.581483. URL: http://doi.acm.org/10.1145/583852.581483.

[18] Daniel Frampton et al. “Demystifying Magic: High-level Low-level Programming”. In: *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE ’09. Washington, DC, USA: ACM, 2009, pp. 81–90. ISBN: 978-1-60558-375-4. DOI: 10.1145/1508293.1508305. URL: http://doi.acm.org/10.1145/1508293.1508305 (cit. on pp. 6, 7).

[19] 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.

[20] Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. “Free-Me: A Static Analysis for Automatic Individual Object Reclamation”. In: *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’06. Ottawa, Ontario, Canada: ACM, 2006, pp. 364–375. ISBN: 1-59593-320-4. DOI: 10.1145/1133981.1134024. URL: http://doi.acm.org/10.1145/1133981.1134024.

[21] 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. 11).

[22] 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. 7, 11).

[23] David Hemmendinger. “Syntax, Semantics, and Pragmatics”. In: *Encyclopedia of Computer Science*. Chichester, UK: John Wiley and Sons Ltd., pp. 1737–1738. ISBN: 0-470-86412-5. URL: http://dl.acm.org.ezproxy1.allegheny.edu:2048/citation.cfm?id=1074100.1074848 (cit. on p. 9).

[24] Fritz Henglein, Henning Makholm, and Henning Niss. “A Direct Approach to Control-flow Sensitive Region-based Memory Management”. In: *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP ’01. Florence, Italy: ACM, 2001, pp. 175–186. ISBN: 1-58113-388-X. DOI: 10.1145/773184.773203. URL: http://doi.acm.org/10.1145/773184.773203.

[25] 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 pp. 2, 6).

[26] Tony Hoare. “Null references: The billion dollar mistake”. In: *Presentation at QCon London* (2009) (cit. on p. 12).

[27] Paul Hudak and Joseph H Fasel. "A gentle introduction to Haskell". In: *ACM Sigplan Notices* 27.5 (1992), pp. 1–52 (cit. on pp. 10, 13).

[28] 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. 9).

[29] 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, 10).

[30] John Hughes. "Why functional programming matters". In: *The Computer Journal* 32.2 (1989), pp. 98–107 (cit. on p. 9).

[31] Graham Hutton and Erik Meijer. *Monadic parser combinators*. Tech. rep. NOTTCS-TR-96-4. 1996. URL: `http://eprints.nottingham.ac.uk/237/`.

[32] "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`.

[33] ISO. *C11 Standard*. ISO/IEC 9899:2011 (cit. on p. 2).

[34] ISO. *C99 Standard*. ISO/IEC 9899:1999 (cit. on p. 2).

[35] Simon L Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003 (cit. on pp. 1, 10, 13).

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

[37] Oleg Kiselyov and Chung-chieh Shan. "Lightweight Monadic Regions". In: *SIGPLAN Not.* 44.2 (Sept. 2008), pp. 1–12. ISSN: 0362-1340. DOI: `10.1145/1543134.1411288`. URL: `http://doi.acm.org/10.1145/1543134.1411288`.

[38] 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 p. 10).

[39] Chris Lattner. "LLVM and Clang: Next generation compiler technology". In: *The BSD Conference*. 2008, pp. 1–2 (cit. on pp. 2, 7).

[40] 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 p. 7).

[41] Daan Leijen and Erik Meijer. "Parsec: Direct style monadic parser combinators for the real world". In: (2002).

[42] 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`.

[43] 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 p. 10).

[44] Luc Maranget. "Warnings for pattern matching". In: *Journal of Functional Programming* 17.03 (2007), pp. 387–421 (cit. on p. 10).

[45] 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`.

[46] 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. 7, 11, 12).

[47] Clemens Mayer et al. "An Empirical Study of the Influence of Static Type Systems on the Usability of Undocumented Software". In: *SIGPLAN Not.* 47.10 (Oct. 2012), pp. 683–702. ISSN: 0362-1340. DOI: `10.1145/2398857.2384666`. URL: `http://doi.acm.org/10.1145/2398857.2384666` (cit. on pp. 8, 9).

[48] John C. Mitchell. "Types, Theory of". In: *Encyclopedia of Computer Science*. Chichester, UK: John Wiley and Sons Ltd., pp. 1806–1810. ISBN: 0-470-86412-5. URL: `http://dl.acm.org/citation.cfm?id=1074100.1074885` (cit. on p. 8).

[49] Adriaan Moors, Frank Piessens, and Martin Odersky. *Parser combinators in Scala*. Tech. rep. Katholieke Universiteit Leuven, 2008.

[50] 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).

[51] Peter Norvig. "Teach yourself programming in ten years". In: *URL http://norvig.com/21-days. html# answers* (2001).

[52] Martin Odersky et al. *An overview of the Scala programming language*. Tech. rep. 2004 (cit. on pp. 1, 10, 13).

[53] Martin Odersky et al. *The Scala language specification*. 2004 (cit. on pp. 1, 10, 13).

[54] 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 pp. 5, 10).

[55] 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. 2, 7).

[56] Eric S Raymond. "How to become a hacker". In: *Database and Network Journal* 33.2 (2003), pp. 8–9 (cit. on p. 8).

[57] Konstantinos Sagonas. "Using Static Analysis to Detect Type Errors and Concurrency Defects in Erlang Programs". In: *Proceedings of the 10th International Conference on Functional and Logic Programming*. FLOPS'10. Sendai, Japan: Springer-Verlag, 2010, pp. 13–18. ISBN: 3-642-12250-7, 978-3-642-12250-7. DOI: `10.1007/978-3-642-12251-4_2`. URL: `http://dx.doi.org/10.1007/978-3-642-12251-4_2` (cit. on p. 8).

[58] 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, 5–7).

[59] Christian Skalka and Scott Smith. "Static Enforcement of Security with Types". In: *SIGPLAN Not.* 35.9 (Sept. 2000), pp. 34–45. ISSN: 0362-1340. DOI: `10.1145/357766.351244`. URL: `http://doi.acm.org/10.1145/357766.351244` (cit. on p. 8).

[60] 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`.

[61] 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. 12).

[62] 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 p. 11).

[63] Robert E Strom and Shaula Yemini. "Typestate: A programming language concept for enhancing software reliability". In: *Software Engineering, IEEE Transactions on* 1 (1986), pp. 157–171.

[64] Gerry Sussman, Harold Abelson, and Julie Sussman. *Structure and interpretation of computer programs*. MIT Press, Cambridge, Mass, 1983 (cit. on pp. 8, 9).

[65] S Doaitse Swierstra. "Combinator parsers: From toys to tools". In: *Electronic Notes in Theoretical Computer Science* 41.1 (2001), pp. 38–59.

[66] 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. 10).

[67] 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 p. 7).

[68] Luke VanderHart and Stuart Sierra. “Macros and Metaprogramming”. In: *Practical Clojure*. Springer, 2010, pp. 167–178 (cit. on p. 9).

[69] 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.

[70] David Wheeler and Alan Gloria. *Sweet-expressions (t-expressions)*. Tech. rep. SRFI-110. 2006. URL: http://srfi.schemers.org/srfi-110/srfi-110.html.

[71] 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. 9).