

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

Copyright © 2015
Hawk Weisman
All rights reserved

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

1	Introduction	1
1.1	Current State of the Art	2
1.2	Goals of the Project	2
1.3	Thesis Outline	3
2	Background and Rationale	4
2.1	Memory Management	4
3	Design Considerations	5
3.1	Design Goals	5
3.2	Characteristics	7
3.3	Experiments	8
3.4	Threats to Validity	8
4	Implementation	9
4.1	Parsing and Syntactic Analysis	9
4.2	Semantic Analysis	9
4.3	Code Generation	9
5	Evaluation	10
6	Discussion and Future Work	11
6.1	Summary of Results	11
6.2	Future Work	11
6.3	Conclusion	11
A	The Mnemosyne Programming Language: An Abridged Description	12
A.1	Introduction	12
A.1.1	Syntactic Notation	12
A.2	Program Structure	13
A.3	Lexical Syntax	13
A.4	Expressions	15
A.4.1	Dereference and Unwrapping Expressions	15
A.4.2	Modules	15
A.5	Definitions	16

A.5.1	Attributes	16
B	Manganese Source Code	17
B.1	Mnemosyne Core Crate	17
B.2	Mnemosyne Parser Crate	44
B.3	Manganese Application Crate	57
	Bibliography	59

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 language¹, we should note that individuals with such wizardly skill² 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 [28, 34], Scala [51, 52], 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 [49].

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 [49, 57]. This is in contrast

¹And inversely, that a terrible programmer can write bad code even in a good language

²Or 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 [35], which first appeared in 1972. While there have been revisions to the C standard in recent years [32, 33] and new C compilers, such as `clang` [38], 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, 54, 57].

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 [57]. 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, 24].

1.2 Goals of the Project

Mnemosyne³ is a new programming language intended for systems programming. Mnemosyne is intended to reconcile the requirements of low-level systems implementation with the

³Named 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.

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 evaluations 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 [53]

ALAN PERLIS

3.1 Design Goals

As is true for all software systems, when implementing a new programming language, The following characteristics are major design goals for Mnemosyne:

1. Support for systems programming
2. Safety
3. Performance
4. Support for quality programming
5. 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. [57, 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 strategies¹” [57, 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 [24] 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.

¹ Viz. garbage collection

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.

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 intended 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 [47].

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 [46, 47].

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 [58] and concurrency defects [56] 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 and modifying code less time-consuming [46]. 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.

3.3 Experiments

3.4 Threats to Validity

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

¹Pronounced “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-compliant 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-compliant Mnemosyne compiler should ensure compatibility 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.

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

$\langle token \rangle \rightarrow \langle lexeme \rangle \mid \langle atmosphere \rangle$

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

$\langle \text{sigil} \rangle \rightarrow '@' | \&' | '*' | '$' | '?'$
 $\langle \text{delimiter} \rangle \rightarrow '(' | ')' | \{ | \}$
 $\langle \text{identifier} \rangle \rightarrow \langle \text{initial} \rangle \langle \text{subsequent} \rangle^*$
 $\langle \text{initial} \rangle \rightarrow \langle \text{letter} \rangle | \langle \text{special initial} \rangle$
 $\langle \text{subsequent} \rangle \rightarrow \langle \text{letter} \rangle | \langle \text{number} \rangle | \langle \text{special subsequent} \rangle$
 $\langle \text{letter} \rangle \rightarrow 'a' | 'b' | 'c' | \dots | 'z'$
 $\quad \quad \quad | 'A' | 'B' | 'C' | \dots | 'Z'$
 $\langle \text{number} \rangle \rightarrow '0' | '1' | \dots | '9'$
 $\langle \text{special initial} \rangle \rightarrow '+' | '-' | '*' | '<' | '>' | '='$
 $\quad \quad \quad | '!' | ':' | '%' | '^'$
 $\langle \text{special subsequent} \rangle \rightarrow \langle \text{special initial} \rangle | '' | '_'$
 $\langle \text{keyword} \rangle \rightarrow \text{'and' | 'begin' | 'borrow' | 'case' | 'cond'}$
 $\quad \quad \quad | \text{'class' | 'data' | 'define' | 'defn' | 'def'}$
 $\quad \quad \quad | \text{'delay' | 'do' | 'else' | 'if' | 'instance'}$
 $\quad \quad \quad | \text{'impl' | 'lambda' | 'let' | 'let*' | 'letrec'}$
 $\quad \quad \quad | \text{'mod' | 'or' | 'quasiquote' | 'quote' | 'ref'}$
 $\quad \quad \quad | \text{'set!' | 'struct' | 'trait' | 'type' | 'typeclass'}$
 $\quad \quad \quad | \text{'union' | 'unquote' | 'unquote-splicing'}$
 $\quad \quad \quad | \langle \text{lambda} \rangle | \langle \text{arrow} \rangle | \langle \text{fat arrow} \rangle$
 $\quad \quad \quad | \langle \text{builtin type} \rangle$
 $\quad \quad \quad | '|' | '->' | '=>' | ','$
 $\langle \text{builtin type} \rangle \rightarrow \text{'i8' | 'i16' | 'i32' | 'i64' | 'int'}$
 $\quad \quad \quad | \text{'u8' | 'ui64' | 'u32' | 'u64' | 'uint'}$
 $\quad \quad \quad | \text{'f32' | 'f64' | 'float' | 'double'}$
 $\quad \quad \quad | \text{'bool' | 'string'}$
 $\langle \text{atmosphere} \rangle \rightarrow \langle \text{whitespace} \rangle | \langle \text{comment} \rangle$
 $\langle \text{whitespace} \rangle \rightarrow \langle \text{space} \rangle$
 $\quad \quad \quad | \langle \text{tab} \rangle \text{ (U+0009)}$
 $\quad \quad \quad | \langle \text{linefeed} \rangle \text{ (U+000A)}$
 $\quad \quad \quad | \langle \text{carriage return} \rangle \text{ (U+000D)}$
 $\langle \text{comment} \rangle \rightarrow ';' \langle \text{any} \rangle^* \langle \text{line ending} \rangle$
 $\quad \quad \quad | '#' \langle \text{any} \rangle^* '#'$

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.

$$\begin{aligned}
 \langle expr \rangle &\rightarrow \langle s\text{-}expr \rangle \mid \langle i\text{-}expr \rangle \mid \langle c\text{-}expr \rangle \mid \langle n\text{-}expr \rangle \\
 &\quad \mid \langle deref\ expr \rangle \mid \langle unwrap\ expr \rangle \mid \langle pointer\ expr \rangle \\
 &\quad \mid \langle literal \rangle \\
 \langle s\text{-}expr \rangle &\rightarrow '(' \langle operator \rangle \langle expr \rangle^* ')', \\
 \langle c\text{-}expr \rangle &\rightarrow '\{ ' \langle expr \rangle \langle operator \rangle \langle c\text{-}expr\ body \rangle^+ '\}', \\
 \langle c\text{-}expr\ body \rangle &\rightarrow \langle expr \rangle \langle operator \rangle \\
 &\quad \mid \langle expr \rangle \\
 \langle n\text{-}expr \rangle &\rightarrow \langle access \rangle '(' \langle expr \rangle^* ')', \\
 &\quad \mid \langle access \rangle '.' \langle identifier \rangle \\
 &\quad \mid \langle access \rangle '.' \langle n\text{-}expr \rangle \\
 \langle access \rangle &\rightarrow \langle deref\text{-}expr \rangle \mid \langle identifier \rangle
 \end{aligned}$$

A.4.1 Dereference and Unwrapping Expressions

$$\begin{aligned}
 \langle deref\ expr \rangle &\rightarrow '(' '\$ ' \langle expr \rangle ')', \\
 &\quad \mid '\$ ' \langle expr \rangle \\
 \langle unwrap\ expr \rangle &\rightarrow '(' '?' \langle expr \rangle \langle expr \rangle ')', \\
 &\quad \mid '(' '?' \langle expr \rangle ')', \\
 &\quad \mid '?' \langle expr \rangle \\
 \langle pointer\ expr \rangle &\rightarrow '(' \langle pointer\ sigil \rangle \langle expr \rangle ')', \\
 &\quad \mid \langle pointer\ sigil \rangle \langle expr \rangle \\
 \langle pointer\ sigil \rangle &\rightarrow '&' \mid '@' \mid '*'
 \end{aligned}$$

A.4.2 Modules

$$\begin{aligned}
 \langle program \rangle &\rightarrow \langle module \rangle^+ \\
 \langle module \rangle &\rightarrow \langle module\text{-}def \rangle \langle definition \rangle^* \\
 \langle module\text{-}def \rangle &\rightarrow '(' 'mod' \langle identifier \rangle \langle exports\text{-}clause \rangle ')',
 \end{aligned}$$

$$\langle \text{exports-clause} \rangle \rightarrow '(\text{'exports' } \langle \text{identifier} \rangle + \text{'})'$$

$$\quad \quad \quad | \quad \varepsilon$$

A.5 Definitions

A.5.1 Attributes

$$\langle \text{function-attrs} \rangle \rightarrow \text{'\#(' } \langle \text{function-attr} \rangle^* \text{'})'}$$

$$\langle \text{function-attr} \rangle \rightarrow \langle \text{any-attr} \rangle$$

$$\quad \quad \quad | \quad \text{'cold'}$$

$$\quad \quad \quad | \quad \langle \text{inline} \rangle$$

$$\langle \text{inline} \rangle \rightarrow \text{'inline'}$$

$$\quad \quad \quad | \quad \text{'inline-always'}$$

$$\langle \text{data-attr} \rangle \rightarrow \langle \text{any-attr} \rangle$$

$$\quad \quad \quad | \quad \text{'(' 'as' } \langle \text{as-attr} \rangle + \text{'})'}$$

$$\quad \quad \quad | \quad \text{'as' '(' } \langle \text{as-attr} \rangle + \text{'})'}$$

$$\langle \text{as-attr} \rangle \rightarrow \text{'C' } | \text{'packed'}$$

$$\quad \quad \quad | \quad \text{'u8' } | \text{'u16' } | \text{'u32' } | \text{'u64'}$$

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

```
1  //
2  // Mnemosyne: a functional systems programming language.
3  // (c) 2015 Hawk Weisman
4  //
5  // Mnemosyne is released under the MIT License. Please refer to
6  // the LICENSE file at the top-level directory of this distribution
7  // or at https://github.com/hawkw/mnemosyne/.
8  //
9  #![crate_name = "mnemosyne"]
10 #![crate_type = "lib"]
11 #![feature(rustc_private)]
12 #![feature(static_recursion)]
13 #![feature(box_syntax, box_patterns)]
14
15 //! # Mnemosyne core
16 //!
17 //! This crate contains the core Mnemosyne programming language components.
18 //! This includes the mnemosyne abstract syntax tree ('semantic::ast'),
19 //! functions for performing semantic analysis ('semantic'), functions
20 //! for compiling abstract syntax trees to LLVM bytecode ('compile'), and
21 //! assorted utility code such as a positional reference type and a
22 //! 'ForkTable' data structure for use as a symbol table.
23 //!
24 //! The Mnemosyne parser is contained in a separate crate in order to improve
```

```

25  //! compile times.
26
27  extern crate rustc;
28  extern crate libc;
29  extern crate combine;
30  extern crate iron_llvm;
31  extern crate llvm_sys;
32  #[macro_use] extern crate itertools;
33
34  use rustc::lib::llvm::{LLVMVersionMajor, LLVMVersionMinor};
35
36  use std::fmt::Debug;
37
38  include!(concat!(env!("OUT_DIR"), "/gen.rs"));
39
40  /// Returns the Mnemosyne version as a String
41  pub fn mnemosyne_version() -> String {
42      format!("Mnemosyne {}", env!("CARGO_PKG_VERSION"))
43  }
44
45  /// Macro for formatting an internal compiler error panic.
46  ///
47  /// This should be used instead of the Rust standard library's 'panic!()'
48  /// macro in the event of an unrecoverable internal compiler error.
49  #[macro_export]
50  macro_rules! ice {
51      ($msg:expr) => (
52          panic!( "[internal error] {}\n \
53                  [internal error] Something has gone horribly wrong.\n \
54                  [internal error] Please contact the Mnemosyne implementors.\n\
55                  {}, {}"
56                  , $msg
57                  , $crate::mnemosyne_version(), $crate::llvm_version()
58              )
59      );
60      ($fmt:expr, $($arg:tt)+) => (
61          panic!( "[internal error] {}\n \
62                  [internal error] Something has gone horribly wrong.\n \
63                  [internal error] Please contact the Mnemosyne implementors.\n\
64                  {}, {}"
65                  , format_args!($fmt, $($arg)+)
66                  , $crate::mnemosyne_version(), $crate::llvm_version()
67              )
68      )
69  }
70
71  pub mod position;
72  pub mod semantic;
73  pub mod compile;
74  pub mod forktable;
75  pub mod chars;
76  pub mod errors;

```

```

77
78 pub use semantic::ast;

```

chars.rs

```

1  /// Unicode code point for the lambda character
2  pub const LAMBDA: &'static str    = "\u{03bb}";
3  /// Unicode code point for the arrow character
4  pub const ARROW: &'static str    = "\u{8594}";
5  /// Unicode code point for the fat arrow (typeclass) character.
6  pub const FAT_ARROW: &'static str = "\u{8685}";
7
8  pub const ALPHA_EXT: &'static str = "+-*/<=>!:$%_~";
9  pub const OPS: &'static str      = "+-*/|=<>";

```

errors.rs

```

1  //
2  // Mnemosyne: a functional systems programming language.
3  // (c) 2015 Hawk Weisman
4  //
5  // Mnemosyne is released under the MIT License. Please refer to
6  // the LICENSE file at the top-level directory of this distribution
7  // or at https://github.com/hawkw/mnemosyne/.
8  //
9
10 use std::fmt::{ Display, Debug };
11
12 /// Mnemosyne error handling
13
14 /// Wraps Option/Result with an 'expect_ice()' method.
15 ///
16 /// The 'expect_ice()' method functions similarly to the standard library's
17 /// 'expect()', but with the custom Mnemosyne internal compiler error message.
18 pub trait ExpectICE<T> {
19     fn expect_ice(self, msg: &str) -> T;
20 }
21
22 impl<T> ExpectICE<T> for Option<T> {
23     /// Unwraps an option, yielding the content of a 'Some'
24     ///
25     /// # Panics
26     ///
27     /// Panics using the Mnemosyne internal compiler error formatter
28     /// if the value is a 'None', with a custom panic message
29     /// provided by 'msg'.
30     ///
31     /// # Examples
32     ///
33     /// ```ignore
34     /// # use mnemosyne::errors::ExpectICE;
35     /// let x = Some("value");

```



```

36     /// assert_eq!(x.expect_ice("the world is ending"), "value");
37     /// '''
38     ///
39     /// '''ignore
40     /// # use mmemosyne::errors::ExpectICE;
41     /// let x: Option<&str> = None;
42     /// x.expect_ice("the world is ending");
43     /// '''
44     #[inline]
45     fn expect_ice(self, msg: &str) -> T {
46         match self { Some(thing) => thing
47                     , None       => ice!(msg)
48                     }
49     }
50 }
51
52 impl<T, E> ExpectICE<T> for Result<T, E>
53 where E: Debug {
54
55     /// Unwraps a result, yielding the content of an 'Ok'.
56     ///
57     /// Panics using the Mmemosyne internal compiler error formatter
58     /// if the value is an 'Err', with a panic message including the
59     /// passed message, and the content of the 'Err'.
60     ///
61     /// # Examples
62     /// '''ignore
63     /// # use mmemosyne::errors::ExpectICE;
64     /// let x: Result<u32, &str> = Err("emergency failure");
65     /// x.expect_ice("Testing expect");
66     /// '''
67     #[inline]
68     fn expect_ice(self, msg: &str) -> T {
69         match self { Ok(t) => t
70                     , Err(e) => ice!("{}", ":{:?}", msg, e)
71                     }
72     }
73 }
74
75 /// Wraps Option/Result with an 'unwrap_ice()' method.
76 ///
77 /// The 'unwrap_ice()' method functions similarly to the standard library's
78 /// 'unwrap()', but with the custom Mmemosyne internal compiler error message.
79 pub trait UnwrapICE<T> {
80     fn unwrap_ice(self) -> T;
81 }
82
83 impl<T> UnwrapICE<T> for Option<T> {
84     /// Moves the value 'v' out of the 'Option<T>' if it is 'Some(v)'.
85     ///
86     /// Unlike the standard library's 'unwrap()', this uses the Mmemosyne
87     /// internal compiler error panic formatter.

```

```

88     ///
89     /// # Panics
90     ///
91     /// Panics if the self value equals 'None'.
92     ///
93     /// # Safety note
94     ///
95     /// In general, because this function may panic, its use is discouraged.
96     /// Instead, prefer to use pattern matching and handle the 'None'
97     /// case explicitly.
98     ///
99     /// # Examples
100    ///
101    /// ```ignore
102    /// # use mnemosyne::errors::UnwrapICE;
103    /// let x = Some("air");
104    /// assert_eq!(x.unwrap_ice(), "air");
105    /// ```
106    ///
107    /// ```ignore
108    /// # use mnemosyne::errors::UnwrapICE;
109    /// let x: Option<&str> = None;
110    /// assert_eq!(x.unwrap_ice(), "air"); // fails
111    /// ```
112    #[inline]
113    fn unwrap_ice(self) -> T {
114        match self { Some(thing) => thing
115                    , None =>
116                        ice!("called 'Option::unwrap()' on a 'None' value")
117                    }
118    }
119 }
120
121 impl<T, E> UnwrapICE<T> for Result<T, E>
122 where E: Display {
123     /// Unwraps a result, yielding the content of an 'Ok'.
124     ///
125     /// Unlike the standard library's 'unwrap()', this uses the Mnemosyne
126     /// internal compiler error panic formatter.
127     ///
128     /// # Panics
129     ///
130     /// Panics if the value is an 'Err', with a panic message provided by the
131     /// 'Err's value.
132     ///
133     /// # Examples
134     ///
135     /// ```ignore
136     /// # use mnemosyne::errors::UnwrapICE;
137     /// let x: Result<u32, &str> = Ok(2);
138     /// assert_eq!(x.unwrap_ice(), 2);
139     /// ```

```

```

140     ///
141     /// '''ignore
142     /// # use mnemosyne::errors::UnwrapICE;
143     /// let x: Result<u32, &str> = Err("emergency failure");
144     /// x.unwrap_ice(); // panics
145     /// '''
146     #[inline]
147     fn unwrap_ice(self) -> T {
148         match self { Ok(t) => t
149                     , Err(e) => ice!("{}", e)
150                     }
151     }
152 }
153 //
154 // impl<T, E> UnwrapICE<T> for Result<T, E>
155 // where E: Debug {
156 //     /// Unwraps a result, yielding the content of an 'Ok'.
157 //     ///
158 //     /// Unlike the standard library's 'unwrap()', this uses the Mnemosyne
159 //     /// internal compiler error panic formatter.
160 //     ///
161 //     /// # Panics
162 //     ///
163 //     /// Panics if the value is an 'Err', with a panic message provided by the
164 //     /// 'Err's value.
165 //     ///
166 //     /// # Examples
167 //     ///
168 //     /// '''
169 //     /// # use mnemosyne::errors::UnwrapICE;
170 //     /// let x: Result<u32, &str> = Ok(2);
171 //     /// assert_eq!(x.unwrap_ice(), 2);
172 //     /// '''
173 //     ///
174 //     /// '''{.should_panic}
175 //     /// # use mnemosyne::errors::UnwrapICE;
176 //     /// let x: Result<u32, &str> = Err("emergency failure");
177 //     /// x.unwrap_ice(); // panics with 'emergency failure'
178 //     /// '''
179 //     #[inline]
180 //     fn unwrap_ice(self) -> T {
181 //         match self {
182 //             Ok(t) => t
183 //             , Err(e) =>
184 //                 ice!("called 'Result::unwrap()' on an 'Err' value: {:?}", e)
185 //         }
186 //     }
187 // }
188
189 #[cfg(test)]
190 mod tests {
191     use super::*;

```

```

192
193     #[test]
194     fn test_option_expect_ok() {
195         let x = Some("value");
196         assert_eq!(x.expect_ice("the world is ending"), "value");
197     }
198
199     #[test]
200     #[should_panic]
201     fn test_option_expect_panic() {
202         let x: Option<&str> = None;
203         x.expect_ice("the world is ending");
204     }
205
206     #[test]
207     #[should_panic]
208     fn test_result_expect_panic() {
209         let x: Result<u32, &str> = Err("emergency failure");
210         x.expect_ice("Testing expect");
211     }
212
213     #[test]
214     fn test_option_unwrap_ok() {
215         let x = Some("air");
216         assert_eq!(x.unwrap_ice(), "air");
217     }
218
219     #[test]
220     #[should_panic]
221     fn test_option_unwrap_panic() {
222         let x: Option<&str> = None;
223         assert_eq!(x.unwrap_ice(), "air"); // fails
224     }
225
226     #[test]
227     fn test_result_unwrap_ok() {
228         let x: Result<u32, &str> = Ok(2);
229         assert_eq!(x.unwrap_ice(), 2);
230     }
231
232     #[test]
233     #[should_panic]
234     fn test_result_unwrap_panic() {
235         let x: Result<u32, &str> = Err("emergency failure");
236         x.unwrap_ice(); // panics
237     }
238 }

```

forktable.rs

```

1 //
2 // Mnemosyne: a functional systems programming language.

```

```

3  // (c) 2015 Hawk Weisman
4  //
5  // Mnemosyne is released under the MIT License. Please refer to
6  // the LICENSE file at the top-level directory of this distribution
7  // or at https://github.com/hawkw/mnemosyne/.
8  //
9
10 use ::errors::ExpectICE;
11
12 use std::collections::{HashMap, HashSet};
13 use std::collections::hash_map::{Keys, Values};
14 use std::hash::Hash;
15 use std::borrow::Borrow;
16 use std::ops;
17
18 /// An associative map data structure for representing scopes.
19 ///
20 /// A 'ForkTable' functions similarly to a standard associative map
21 /// data structure (such as a 'HashMap'), but with the ability to
22 /// fork children off of each level of the map. If a key exists in any
23 /// of a child's parents, the child will 'pass through' that key. If a
24 /// new value is bound to a key in a child level, that child will overwrite
25 /// the previous entry with the new one, but the previous 'key' -> 'value'
26 /// mapping will remain in the level it is defined. This means that the parent
27 /// level will still provide the previous value for that key.
28 ///
29 /// This is an implementation of the ForkTable data structure for
30 /// representing scopes. The ForkTable was initially described by
31 /// Max Clive. This implementation is based primarily by the Scala
32 /// reference implementation written by Hawk Weisman for the Decaf
33 /// compiler, which is available [here](https://github.com/hawkw/decaf/blob/master/src/main/scala)
34 #[derive(Debug, Clone)]
35 pub struct ForkTable<'a, K, V>
36 where K: Eq + Hash
37     , K: 'a
38     , V: 'a
39 {
40     table: HashMap<K, V>
41     , whiteouts: HashSet<K>
42     , parent: Option<&'a ForkTable<'a, K, V>>
43     , level: usize
44 }
45
46 impl<'a, K, V> ForkTable<'a, K, V>
47 where K: Eq + Hash
48 {
49
50     /// Returns a reference to the value corresponding to the key.
51     ///
52     /// If the key is defined in this level of the table, or in any
53     /// of its' parents, a reference to the associated value will be
54     /// returned.

```

```

55  ///
56  /// The key may be any borrowed form of the map's key type, but
57  /// 'Hash' and 'Eq' on the borrowed form must match those for
58  /// the key type.
59  ///
60  /// # Arguments
61  ///
62  /// + 'key' - the key to search for
63  ///
64  /// # Return Value
65  ///
66  /// + 'Some(&V)' if an entry for the given key exists in the
67  ///   table, or 'None' if there is no entry for that key.
68  ///
69  /// # Examples
70  ///
71  /// ```ignore
72  /// # use mmemosyne::forktable::ForkTable;
73  /// let mut table: ForkTable<isize,&str> = ForkTable::new();
74  /// assert_eq!(table.get(&1), None);
75  /// table.insert(1, "One");
76  /// assert_eq!(table.get(&1), Some(&"One"));
77  /// assert_eq!(table.get(&2), None);
78  /// ```
79  /// ```ignore
80  /// # use mmemosyne::forktable::ForkTable;
81  /// let mut level_1: ForkTable<isize,&str> = ForkTable::new();
82  /// level_1.insert(1, "One");
83  ///
84  /// let mut level_2: ForkTable<isize,&str> = level_1.fork();
85  /// assert_eq!(level_2.get(&1), Some(&"One"));
86  /// ```
87  pub fn get<Q: ?Sized>(&self, key: &Q) -> Option<&V>
88  where K: Borrow<Q>
89        , Q: Hash + Eq
90  {
91      if self.whiteouts.contains(key) {
92          None
93      } else {
94          self.table
95              .get(key)
96              .or(self.parent
97                  .map_or(None, |ref parent| parent.get(key))
98                  )
99      }
100  }
101
102  /// Returns a mutable reference to the value corresponding to the key.
103  ///
104  /// If the key is defined in this level of the table, a reference to the
105  /// associated value will be returned.
106  ///

```

```

107  /// Note that only keys defined in this level of the table can be accessed
108  /// as mutable. This is because otherwise it would be necessary for each
109  /// level of the table to hold a mutable reference to its parent.
110  ///
111  /// The key may be any borrowed form of the map's key type, but
112  /// 'Hash' and 'Eq' on the borrowed form *must* match those for
113  /// the key type.
114  ///
115  /// # Arguments
116  ///
117  /// + 'key' - the key to search for
118  ///
119  /// # Return Value
120  ///
121  /// + 'Some(&mut V)' if an entry for the given key exists in the
122  ///   table, or 'None' if there is no entry for that key.
123  ///
124  /// # Examples
125  ///
126  /// ```ignore
127  /// # use mmemosyne::forktable::ForkTable;
128  /// let mut table: ForkTable<isize,&str> = ForkTable::new();
129  /// assert_eq!(table.get_mut(&1), None);
130  /// table.insert(1, "One");
131  /// assert_eq!(table.get_mut(&1), Some(&mut "One"));
132  /// assert_eq!(table.get_mut(&2), None);
133  /// ```
134  /// ```ignore
135  /// # use mmemosyne::forktable::ForkTable;
136  /// let mut level_1: ForkTable<isize,&str> = ForkTable::new();
137  /// level_1.insert(1, "One");
138  ///
139  /// let mut level_2: ForkTable<isize,&str> = level_1.fork();
140  /// assert_eq!(level_2.get_mut(&1), None);
141  /// ```
142  pub fn get_mut<Q: ?Sized>(&mut self, key: &Q) -> Option<&mut V>
143  where K: Borrow<Q>
144        , Q: Hash + Eq
145  {
146      self.table.get_mut(key)
147  }
148
149
150  /// Removes a key from the map, returning the value at the key if
151  /// the key was previously in the map.
152  ///
153  /// If the removed value exists in a lower level of the table,
154  /// it will be whited out at this level. This means that the entry
155  /// will be 'removed' at this level and this table will not provide
156  /// access to it, but the mapping will still exist in the level where
157  /// it was defined. Note that the key will not be returned if it is
158  /// defined in a lower level of the table.

```

```

159  ///
160  /// The key may be any borrowed form of the map's key type, but
161  /// 'Hash' and 'Eq' on the borrowed form *must* match those for
162  /// the key type.
163  ///
164  /// # Arguments
165  ///
166  /// + 'key' - the key to remove
167  ///
168  /// # Return Value
169  ///
170  /// + 'Some(V)' if an entry for the given key exists in the
171  ///   table, or 'None' if there is no entry for that key.
172  ///
173  /// # Examples
174  /// ```ignore
175  /// # use mmemosyne::forktable::ForkTable;
176  /// let mut table: ForkTable<isize,&str> = ForkTable::new();
177  /// table.insert(1, "One");
178  ///
179  /// assert_eq!(table.remove(&1), Some("One"));
180  /// assert_eq!(table.contains_key(&1), false);
181  /// ```
182  /// ```ignore
183  /// # use mmemosyne::forktable::ForkTable;
184  /// let mut level_1: ForkTable<isize,&str> = ForkTable::new();
185  /// level_1.insert(1, "One");
186  /// assert_eq!(level_1.contains_key(&1), true);
187  ///
188  /// let mut level_2: ForkTable<isize,&str> = level_1.fork();
189  /// assert_eq!(level_2.chain_contains_key(&1), true);
190  /// assert_eq!(level_2.remove(&1), None);
191  /// assert_eq!(level_2.chain_contains_key(&1), false);
192  /// ```
193  pub fn remove(&mut self, key: &K) -> Option<V>
194  where K: Clone
195  {
196      self.whiteouts.insert(key.clone());
197      self.table.remove(&key)
198  }
199
200  /// Removes a key from this layer's map and whiteouts, so that
201  /// definitions of that key from lower levels are exposed.
202  ///
203  /// Unlike 'ForkTable::remove()', if the removed value exists in a
204  /// lower level of the table, it will NOT be whited out. This means
205  /// that the definition of that entry from lower levels of the table
206  /// will be exposed at this level.
207  ///
208  /// The key may be any borrowed form of the map's key type, but
209  /// 'Hash' and 'Eq' on the borrowed form *must* match those for
210  /// the key type.

```



```

211  ///
212  /// # Arguments
213  ///
214  /// + 'key' - the key to expose
215  ///
216  /// # Return Value
217  ///
218  /// + 'Some(V)' if an entry for the given key exists in the
219  ///    table, or 'None' if there is no entry for that key.
220  ///
221  pub fn expose<Q: ?Sized>(&mut self, key: &Q) -> Option<V>
222  where K: Borrow<Q>
223        , Q: Hash + Eq
224  {
225      self.whiteouts.remove(key);
226      self.table.remove(key)
227  }
228
229  /// Inserts a key-value pair from the map.
230  ///
231  /// If the key already had a value present in the map, that
232  /// value is returned. Otherwise, 'None' is returned.
233  ///
234  /// If the key is currently whited out (i.e. it was defined
235  /// in a lower level of the map and was removed) then it will
236  /// be un-whited out and added at this level.
237  ///
238  /// # Arguments
239  ///
240  /// + 'k' - the key to add
241  /// + 'v' - the value to associate with that key
242  ///
243  /// # Return Value
244  ///
245  /// + 'Some(V)' if a previous entry for the given key exists in the
246  ///    table, or 'None' if there is no entry for that key.
247  ///
248  /// # Examples
249  ///
250  /// Simply inserting an entry:
251  ///
252  /// ```ignore
253  /// # use mnemosyne::forktable::ForkTable;
254  /// let mut table: ForkTable<isize,&str> = ForkTable::new();
255  /// assert_eq!(table.get(&1), None);
256  /// table.insert(1, "One");
257  /// assert_eq!(table.get(&1), Some(&"One"));
258  /// ```
259  ///
260  /// Overwriting the value associated with a key:
261  ///
262  /// ```ignore

```

```

263  /// # use mmemosyne::forktable::ForkTable;
264  /// let mut table: ForkTable<isize,&str> = ForkTable::new();
265  /// assert_eq!(table.get(&1), None);
266  /// assert_eq!(table.insert(1, "one"), None);
267  /// assert_eq!(table.get(&1), Some(&"one"));
268  ///
269  /// assert_eq!(table.insert(1, "One"), Some("one"));
270  /// assert_eq!(table.get(&1), Some(&"One"));
271  /// '''
272  pub fn insert(&mut self, k: K, v: V) -> Option<V> {
273      if self.whiteouts.contains(&k) {
274          self.whiteouts.remove(&k);
275      };
276      self.table.insert(k, v)
277  }
278
279  /// Returns true if this level contains a value for the specified key.
280  ///
281  /// The key may be any borrowed form of the map's key type, but
282  /// 'Hash' and 'Eq' on the borrowed form *must* match those for
283  /// the key type.
284  ///
285  /// # Arguments
286  ///
287  /// + 'k' - the key to search for
288  ///
289  /// # Return Value
290  ///
291  /// + 'true' if the given key is defined in this level of the
292  /// table, 'false' if it does not.
293  ///
294  /// # Examples
295  /// '''ignore
296  /// # use mmemosyne::forktable::ForkTable;
297  /// let mut table: ForkTable<isize,&str> = ForkTable::new();
298  /// assert_eq!(table.contains_key(&1), false);
299  /// table.insert(1, "One");
300  /// assert_eq!(table.contains_key(&1), true);
301  /// '''
302  /// '''ignore
303  /// # use mmemosyne::forktable::ForkTable;
304  /// let mut level_1: ForkTable<isize,&str> = ForkTable::new();
305  /// assert_eq!(level_1.contains_key(&1), false);
306  /// level_1.insert(1, "One");
307  /// assert_eq!(level_1.contains_key(&1), true);
308  ///
309  /// let mut level_2: ForkTable<isize,&str> = level_1.fork();
310  /// assert_eq!(level_2.contains_key(&1), false);
311  /// '''
312  pub fn contains_key<Q: ?Sized>(&self, key: &Q) -> bool
313  where K: Borrow<Q>
314      , Q: Hash + Eq

```

```

315 {
316     !self.whiteouts.contains(key) &&
317     self.table.contains_key(key)
318 }
319
320 /// Returns true if the key is defined in this level of the table, or
321 /// in any of its' parents and is not whited out.
322 ///
323 /// The key may be any borrowed form of the map's key type, but
324 /// 'Hash' and 'Eq' on the borrowed form *must* match those for
325 /// the key type.
326 ///
327 /// # Arguments
328 ///
329 /// + 'k' - the key to search for
330 ///
331 /// # Return Value
332 ///
333 /// + 'true' if the given key is defined in the table,
334 ///   'false' if it does not.
335 ///
336 /// # Examples
337 /// '''ignore
338 /// # use mmemosyne::forktable::ForkTable;
339 /// let mut table: ForkTable<isize,&str> = ForkTable::new();
340 /// assert_eq!(table.chain_contains_key(&1), false);
341 /// table.insert(1, "One");
342 /// assert_eq!(table.chain_contains_key(&1), true);
343 /// '''
344 /// '''ignore
345 /// # use mmemosyne::forktable::ForkTable;
346 /// let mut level_1: ForkTable<isize,&str> = ForkTable::new();
347 /// assert_eq!(level_1.chain_contains_key(&1), false);
348 /// level_1.insert(1, "One");
349 /// assert_eq!(level_1.chain_contains_key(&1), true);
350 ///
351 /// let mut level_2: ForkTable<isize,&str> = level_1.fork();
352 /// assert_eq!(level_2.chain_contains_key(&1), true);
353 /// '''
354 pub fn chain_contains_key<Q:? Sized>(&self, key: &Q) -> bool
355 where K: Borrow<Q>
356     , Q: Hash + Eq
357 {
358     self.table.contains_key(key) ||
359     (!self.whiteouts.contains(key) &&
360     self.parent
361     .map_or(false, |ref p| p.chain_contains_key(key))
362     )
363 }
364
365 /// Forks this table, returning a new 'ForkTable<K,V>'.
366 ///

```

```

367     /// This level of the table will be set as the child's
368     /// parent. The child will be created with an empty backing
369     /// 'HashMap' and no keys whited out.
370     ///
371     /// Note that the new 'ForkTable<K,V>' has a lifetime
372     /// bound ensuring that it will live at least as long as the
373     /// parent 'ForkTable'.
374     pub fn fork(&'a self) -> ForkTable<'a, K, V> {
375         ForkTable { table: HashMap::new()
376                     , whiteouts: HashSet::new()
377                     , parent: Some(self)
378                     , level: self.level + 1
379                     }
380     }
381
382     /// Constructs a new 'ForkTable<K,V>'
383     pub fn new() -> ForkTable<'a, K,V> {
384         ForkTable { table: HashMap::new()
385                     , whiteouts: HashSet::new()
386                     , parent: None
387                     , level: 0
388                     }
389     }
390
391     /// Wrapper for the backing map's 'values()' function.
392     ///
393     /// Provides an iterator visiting all values in arbitrary
394     /// order. Iterator element type is &'b V.
395     pub fn values(&self) -> Values<K, V> { self.table.values() }
396
397     /// Wrapper for the backing map's 'keys()' function.
398     ///
399     /// Provides an iterator visiting all keys in arbitrary
400     /// order. Iterator element type is &'b K.
401     pub fn keys(&self) -> Keys<K, V> { self.table.keys() }
402 }
403
404     /// Allows 'table[&key]' indexing syntax.
405     ///
406     /// This is just a wrapper for 'get(&key)'
407     ///
408     /// '''ignore
409     /// # use mnemosyne::forktable::ForkTable;
410     /// let mut table: ForkTable<isize,&str> = ForkTable::new();
411     /// table.insert(1, "One");
412     /// assert_eq!(table[&1], "One");
413     /// '''
414     impl<'a, 'b, K, Q: ?Sized, V> ops::Index<&'b Q> for ForkTable<'a, K, V>
415     where K: Borrow<Q>
416         , K: Eq + Hash
417         , Q: Eq + Hash
418     {

```

```

419     type Output = V;
420
421     #[inline]
422     fn index(&self, index: &Q) -> &Self::Output {
423         self.get(index)
424             .expect_ice("undefined index")
425     }
426
427 }
428
429 /// Allows mutable 'table[&key]' indexing syntax.
430 ///
431 /// This is just a wrapper for 'get_mut(&key)'
432 ///
433 /// '''ignore
434 /// # use mnemosyne::forktable::ForkTable;
435 /// let mut table: ForkTable<isize,&str> = ForkTable::new();
436 /// table.insert(1, "One");
437 /// table[&1] = "one";
438 /// assert_eq!(table[&1], "one")
439 /// '''
440 impl<'a, 'b, K, Q: ?Sized, V> ops::IndexMut<&'b Q> for ForkTable<'a, K, V>
441 where K: Borrow<Q>
442     , K: Eq + Hash
443     , Q: Eq + Hash
444 {
445     #[inline]
446     fn index_mut(&mut self, index: &Q) -> &mut V {
447         self.get_mut(index)
448             .expect_ice("undefined index")
449     }
450
451 }
452
453 #[cfg(test)]
454 mod tests {
455     use super::ForkTable;
456
457     #[test]
458     fn test_get_defined() {
459         let mut table: ForkTable<isize,&str> = ForkTable::new();
460         assert_eq!(table.get(&1), None);
461         table.insert(1, "One");
462         assert_eq!(table.get(&1), Some(&"One"));
463     }
464
465     #[test]
466     fn test_get_undefined() {
467         let mut table: ForkTable<isize,&str> = ForkTable::new();
468         table.insert(1, "One");
469         assert_eq!(table.get(&2), None);
470     }

```

```

471  #[test]
472  fn test_get_multilevel() {
473      let mut level_1: ForkTable<isize,&str> = ForkTable::new();
474      level_1.insert(1, "One");
475
476      let mut level_2: ForkTable<isize,&str> = level_1.fork();
477      assert_eq!(level_2.get(&1), Some(&"One"));
478  }
479
480  #[test]
481  fn test_get_mut_defined() {
482      let mut table: ForkTable<isize,&str> = ForkTable::new();
483      assert_eq!(table.get_mut(&1), None);
484      table.insert(1, "One");
485      assert_eq!(table.get_mut(&1), Some(&mut "One"));
486  }
487
488  #[test]
489  fn test_get_mut_undefined() {
490      let mut table: ForkTable<isize,&str> = ForkTable::new();
491      table.insert(1, "One");
492      assert_eq!(table.get_mut(&2), None);
493  }
494  #[test]
495  fn test_get_mut_multilevel() {
496      let mut level_1: ForkTable<isize,&str> = ForkTable::new();
497      level_1.insert(1, "One");
498
499      let mut level_2: ForkTable<isize,&str> = level_1.fork();
500      assert_eq!(level_2.get_mut(&1), None);
501  }
502  #[test]
503  fn test_remove_returned() {
504      let mut table: ForkTable<isize,&str> = ForkTable::new();
505      table.insert(1, "One");
506      assert_eq!(table.remove(&1), Some("One"));
507  }
508  #[test]
509  fn test_remove_not_defined_after() {
510      let mut table: ForkTable<isize,&str> = ForkTable::new();
511      table.insert(1, "One");
512      table.remove(&1);
513      assert_eq!(table.get(&1), None);
514  }
515
516  #[test]
517  fn test_remove_multilevel() {
518      let mut level_1: ForkTable<isize,&str> = ForkTable::new();
519      level_1.insert(1, "One");
520      assert_eq!(level_1.contains_key(&1), true);
521
522      let mut level_2: ForkTable<isize,&str> = level_1.fork();

```

```

523         assert_eq!(level_2.chain_contains_key(&1), true);
524         assert_eq!(level_2.remove(&1), None);
525         assert_eq!(level_2.chain_contains_key(&1), false);
526     }
527
528     #[test]
529     fn test_insert_defined_after() {
530         let mut table: ForkTable<isize,&str> = ForkTable::new();
531         assert_eq!(table.get(&1), None);
532         table.insert(1, "One");
533         assert_eq!(table.get(&1), Some(&"One"));
534     }
535
536     #[test]
537     fn test_insert_overwrite() {
538         let mut table: ForkTable<isize,&str> = ForkTable::new();
539         assert_eq!(table.get(&1), None);
540         assert_eq!(table.insert(1, "one"), None);
541         assert_eq!(table.get(&1), Some(&"one"));
542
543         assert_eq!(table.insert(1, "One"), Some(&"one"));
544         assert_eq!(table.get(&1), Some(&"One"));
545     }
546
547     #[test]
548     fn test_contains_key() {
549         let mut table: ForkTable<isize,&str> = ForkTable::new();
550         assert_eq!(table.contains_key(&1), false);
551         table.insert(1, "One");
552         assert_eq!(table.contains_key(&1), true);
553     }
554
555     #[test]
556     fn test_contains_key_this_level_only () {
557         let mut level_1: ForkTable<isize,&str> = ForkTable::new();
558         assert_eq!(level_1.contains_key(&1), false);
559         level_1.insert(1, "One");
560         assert_eq!(level_1.contains_key(&1), true);
561
562         let mut level_2: ForkTable<isize,&str> = level_1.fork();
563         assert_eq!(level_2.contains_key(&1), false);
564     }
565
566     #[test]
567     fn test_chain_contains_key_this_level() {
568         let mut table: ForkTable<isize,&str> = ForkTable::new();
569         assert_eq!(table.chain_contains_key(&1), false);
570         table.insert(1, "One");
571         assert_eq!(table.chain_contains_key(&1), true);
572     }
573
574     #[test]

```

```

575     fn test_contains_key_multilevel() {
576         let mut level_1: ForkTable<isize,&str> = ForkTable::new();
577         assert_eq!(level_1.chain_contains_key(&1), false);
578         level_1.insert(1, "One");
579         assert_eq!(level_1.chain_contains_key(&1), true);
580
581         let mut level_2: ForkTable<isize,&str> = level_1.fork();
582         assert_eq!(level_2.chain_contains_key(&1), true);
583     }
584
585     #[test]
586     fn test_indexing() {
587         let mut table: ForkTable<isize,&str> = ForkTable::new();
588         table.insert(1, "One");
589         assert_eq!(table[&1], "One");
590     }
591
592     #[test]
593     fn test_index_mut() {
594         let mut table: ForkTable<isize,&str> = ForkTable::new();
595         table.insert(1, "One");
596         table[&1] = "one";
597         assert_eq!(table[&1], "one")
598     }
599 }

```

position.rs

```

1  //
2  // Mnemosyne: a functional systems programming language.
3  // (c) 2015 Hawk Weisman
4  //
5  // Mnemosyne is released under the MIT License. Please refer to
6  // the LICENSE file at the top-level directory of this distribution
7  // or at https://github.com/hawkw/mnemosyne/.
8  //
9
10 use std::ops::{Deref, DerefMut};
11 use std::hash;
12 use std::fmt;
13 use std::convert::From;
14
15 use combine::primitives::SourcePosition;
16
17 /// Struct representing a position within a source code file.
18 ///
19 /// This represents positions using 'i32's because that's how
20 /// positions are represented in 'combine' (the parsing library
21 /// that we will use for the Mnemosyne parser). I personally would
22 /// have used 'usize's...
23 #[derive(Copy, Clone, PartialEq, Eq, Debug, PartialOrd, Ord)]
24 pub struct Position { pub col: i32

```



```

25         , pub row: i32
26         , pub raw: i32
27     }
28
29 impl Position {
30
31     /// Create a new 'Position' at the given column and row.
32     #[inline]
33     pub fn new(col: i32, row: i32) -> Self {
34         Position { col: col
35                   , row: row
36                   , raw: col + row
37                 }
38     }
39
40 }
41
42 impl From<SourcePosition> for Position {
43     /// Create a new 'Position' from a 'combine' 'SourcePosition'.
44     ///
45     /// # Example
46     /// ```ignore
47     /// # extern crate combine;
48     /// # extern crate memosyne;
49     /// # use combine::primitives::SourcePosition;
50     /// # use memosyne::position::Position;
51     /// # fn main() {
52     ///     let sp = SourcePosition { column: 1, line: 1 };
53     ///     assert_eq!(Position::from(sp), Position::new(1,1));
54     /// # }
55     /// ```
56     fn from(p: SourcePosition) -> Self { Position::new(p.column, p.line) }
57 }
58
59 impl From<(i32,i32)> for Position {
60     /// Create a new 'Position' from a tuple of i32s.
61     ///
62     /// # Example
63     /// ```ignore
64     /// # use memosyne::position::Position;
65     /// let tuple: (i32,i32) = (1,1);
66     /// assert_eq!(Position::from(tuple), Position::new(1,1));
67     /// ```
68     fn from((col, row): (i32,i32)) -> Self { Position::new(col,row) }
69 }
70
71
72 impl fmt::Display for Position {
73     fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
74         write!(f, "line {}, column {}", self.row, self.col)
75     }
76 }

```

```

77
78 /// A pointer to a value with an associated 'Position'
79 #[derive(Clone, Debug)]
80 pub struct Positional<T> { pub pos: Position
81                           , pub value: T
82                           }
83
84 impl<T> Positional<T> {
85     /// Create a new Positional marker at the given position.
86     pub fn at(col: i32, row: i32, value: T) -> Positional<T> {
87         Positional { pos: Position::new(col, row)
88                     , value: value }
89     }
90
91     pub fn value(&self) -> &T { &self.value }
92 }
93
94
95 impl<T> fmt::Display for Positional<T>
96 where T: fmt::Display {
97     fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
98         write!(f, "{} at {}", self.value, self.pos)
99     }
100 }
101
102 /// A positional pointer is still equal to the underlying
103 /// value even if they have different positions. This is
104 /// important so that we can test that two identifiers are
105 /// the same.
106 impl<T> PartialEq for Positional<T>
107 where T: PartialEq {
108     fn eq(&self, other: &Positional<T>) -> bool {
109         self.value == other.value
110     }
111 }
112
113 /// If two things are equal, then they better have the same
114 /// hash as well. Otherwise there will be sadness.
115 ///
116 /// Hopefully this is Ideologically Correct.
117 impl<T> hash::Hash for Positional<T>
118 where T: hash::Hash {
119     fn hash<H: hash::Hasher>(&self, state: &mut H) {
120         self.value.hash(state)
121     }
122 }
123
124
125 /// This is literally just waving my hands for the compiler.
126 ///
127 /// Hopefully it understands what I mean.
128 impl<T> Eq for Positional<T>

```

```

129 where T: Eq
130     , T: PartialEq
131     {}
132
133 impl<T> Deref for Positional<T> {
134     type Target = T;
135     fn deref(&self) -> &T {
136         &self.value
137     }
138 }
139
140 impl<T> DerefMut for Positional<T> {
141     fn deref_mut(&mut self) -> &mut T {
142         &mut self.value
143     }
144 }
145
146 #[cfg(test)]
147 mod tests {
148     use super::*;
149     use combine::primitives::SourcePosition;
150
151     #[test]
152     fn test_from_sourceposition() {
153         let sp = SourcePosition { column: 1, line: 1 };
154         assert_eq!(Position::from(sp), Position::new(1,1));
155     }
156
157     #[test]
158     fn test_from_tuple() {
159         let tuple: (i32,i32) = (1,1);
160         assert_eq!(Position::from(tuple), Position::new(1,1));
161     }
162 }

```

compile/mod.rs

```

1  //
2  // Mnemosyne: a functional systems programming language.
3  // (c) 2015 Hawk Weisman
4  //
5  // Mnemosyne is released under the MIT License. Please refer to
6  // the LICENSE file at the top-level directory of this distribution
7  // or at https://github.com/hawkw/mnemosyne/.
8  //
9
10 use std::ffi::CString;
11 use std::cmp::Ordering;
12 use std::mem;
13

```

```

14 use libc::c_uint;
15 use llvm_sys::prelude::LLVMValueRef;
16
17 use iron_llvm::core;
18 use iron_llvm::core::types::{ Type
19                               , TypeCtor
20                               , RealTypeCtor
21                               , RealTypeRef
22                               , IntTypeCtor
23                               , IntTypeRef
24                               };
25 use iron_llvm::{LLVMRef, LLVMRefCtor};
26
27 use errors::ExpectICE;
28 use forktable::ForkTable;
29 use position::Positional;
30 use ast::{ Node
31           , Form
32           , DefForm
33           , Ident
34           , Function };
35
36 use semantic::annotations::{ ScopedState
37                             , Scoped
38                             };
39 use semantic::types;
40 use semantic::types::{ Primitive
41                       , Reference
42                       };
43
44 /// Result type for compiling an AST node to LLVM IR
45 ///
46 /// An 'IRResult' contains either a 'ValueRef', if compilation was successful,
47 /// or a 'Positional<String>' containing an error message and the position of
48 /// the line of code which could not be compiled.
49 pub type IRResult = Result<LLVMValueRef, Vec<Positional<String>>>;
50
51 /// Result type for compiling a type to an LLVM 'TypeRef'.
52 pub type TypeResult<T: Type + Sized> = Result<T, Positional<String>>;
53
54 pub type NamedValues<'a> = ForkTable<'a, &'a str, LLVMValueRef>;
55
56 #[inline] fn word_size() -> usize { mem::size_of::<isize>() }
57
58 /// Trait for that which may join in The Great Work
59 pub trait Compile {
60     /// Compile 'self' to an LLVM 'ValueRef'
61     ///
62     /// # Returns:
63     /// - 'Ok' containing a 'ValueRef' if this was compiled correctly.
64     /// - An 'Err' with a vector of error messages containing any
65     /// errors that occurred during compilation.

```

```

66     ///
67     /// # Panics:
68     /// - If something has gone horribly wrong. This does NOT panic if the
69     ///   code could not be compiled because it was incorrect, but it will
70     ///   panic in the event of an internal compiler error.
71     fn to_ir(&self, context: LLVMContext) -> IRResult;
72 }
73
74 /// Trait for type tags that can be translated to LLVM
75 // pub trait TranslateType {
76 //     /// Translate 'self' to an LLVM 'TypeRef'
77 //     ///
78 //     /// # Returns:
79 //     /// - 'Ok' containing a 'TypeRef' if this was compiled correctly.
80 //     /// - An 'Err' with a positional error message in the event of
81 //     ///   a type error.
82 //     ///
83 //     /// # Panics:
84 //     /// - In the event of an internal compiler error (i.e. if a well-formed
85 //     ///   type could not be gotten from LLVM correctly).
86 //     fn translate_type(&self, context: LLVMContext) -> TypeResult;
87 // }
88
89 /// LLVM compilation context.
90 ///
91 /// This is based rather loosely on MIT License code from
92 /// the [iron-kaleidoscope](https://github.com/jauhnien/iron-kaleidoscope)
93 /// tutorial, and from ['librustc_trans'](https://github.com/rust-lang/rust/blob/master/src/librustc_trans)
94 /// from the Rust compiler.
95 pub struct LLVMContext<'a> { llctx: core::Context
96                             , llmod: core::Module
97                             , llbuilder: core::Builder
98                             , named_vals: NamedValues<'a>
99                             }
100
101 /// because we are in the Raw Pointer Sadness Zone (read: unsafe),
102 /// it is necessary that we assert that everything exists.
103 macro_rules! not_null {
104     ($target:expr) => ({
105         let e = $target;
106         if e.is_null() {
107             ice!( "assertion failed: {} returned null!"
108                 , stringify!($target)
109             );
110         } else { e }
111     })
112 }
113
114 /// converts a raw pointer that may be null to an Option
115 /// the compiler will yell about this, claiming that it involves
116 /// an unused unsafe block, but the unsafe block is usually necessary.
117 macro_rules! optionalise {

```

```

118     ($target:expr) => ({
119         let e = unsafe { $target };
120         if e.is_null() {
121             None
122         } else { Some(e) }
123     })
124 }
125
126 macro_rules! try_vec {
127     ($expr:expr) => ({
128         if !$expr.is_empty() {
129             return Err($expr)
130         }
131     })
132 }
133
134 // ----- SEGFAULT EXISTS SOMEWHERE BELOW THIS LINE -----
135
136 //
137 // impl<'a> LLVMContext<'a> {
138 //
139 //     /// Constructs a new LLVM context.
140 //     ///
141 //     /// # Returns:
142 //     ///     - An 'LLVMContext'
143 //     ///
144 //     /// # Panics:
145 //     ///     - If the LLVM C ABI returned a null value for the 'Context',
146 //     ///       'Builder', or 'Module'
147 //     pub fn new(module_name: &str) -> Self {
148 //         LLVMContext { llctx: core::Context::get_global()
149 //             , llmod: core::Module::new(module_name)
150 //             , llbuilder: core::Builder::new()
151 //             , named_vals: NamedValues::new()
152 //         }
153 //     }
154 //
155 //     /// Dump the module's contents to stderr for debugging
156 //     ///
157 //     /// Apparently this is the only reasonable way to get a textual
158 //     /// representation of a 'Module' in LLVM
159 //     pub fn dump(&self) { self.llmod.dump() }
160 //
161 //     pub fn int_type(&self, size: usize) -> IntTypeRef {
162 //         IntTypeRef::get_int_in_context(&self.llctx, size as c_uint)
163 //     }
164 //
165 //     pub fn float_type(&self) -> RealTypeRef {
166 //         RealTypeRef::get_float_in_context(&self.llctx)
167 //     }
168 //     pub fn double_type(&self) -> RealTypeRef {
169 //         RealTypeRef::get_double_in_context(&self.llctx)

```

```

170 // }
171 // pub fn byte_type(&self) -> IntTypeRef {
172 //     IntTypeRef::get_int8_in_context(&self.llctx)
173 // }
174 //
175 //     /// Get any existing declarations for a given function name.
176 //     ///
177 //     /// # Returns:
178 //     ///     - 'Some' if there is an existing previous declaration
179 //     ///       for this function.
180 //     ///     - 'None' if the function has not been declared previously.
181 //     ///
182 //     /// # Panics:
183 //     ///     - If the C string representation for the function name could
184 //     ///       not be created.
185 //     pub fn get_fn(&self, name: &Ident) -> Option<core::FunctionRef> {
186 //         self.llmod.get_function_by_name(name.value.as_ref())
187 //     }
188 // }
189
190 // impl<'a> Compile for Scoped<'a, Form<'a, ScopedState>> {
191 //     fn to_ir(&self, context: LLVMContext) -> IRResult {
192 //         match **self {
193 //             Form::Define(ref form) => unimplemented!()
194 //             , Form::Let(ref form) => unimplemented!()
195 //             , Form::If { .. } => unimplemented!()
196 //             , Form::Call { .. } => unimplemented!()
197 //             , Form::Lambda(ref fun) => unimplemented!()
198 //             , Form::Logical(ref exp) => unimplemented!()
199 //             , Form::Lit(ref c) => unimplemented!()
200 //             , Form::NameRef(ref form) => unimplemented!()
201 //         }
202 //     }
203 // }
204 //
205 // impl<'a> Compile for Scoped<'a, DefForm<'a, ScopedState>> {
206 //     fn to_ir(&self, context: LLVMContext) -> IRResult {
207 //         match **self {
208 //             DefForm::TopLevel { ref name, ref value, .. } =>
209 //                 unimplemented!()
210 //             , DefForm::Function { ref name, ref fun } => {
211 //                 match context.get_fn(name) {
212 //                     Some(previous) => unimplemented!()
213 //                     , None => unimplemented!()
214 //                 }
215 //             }
216 //         }
217 //     }
218 // }
219 //
220 //
221 // impl<'a> Compile for Scoped<'a, Function<'a, ScopedState>> {

```

```

222 //
223 //   fn to_ir(&self, context: LLVMContext) -> IRResult {
224 //       let mut errs: Vec<Positional<String>> = vec![];
225 //       // Check to see if the pattern binds an equivalent number of arguments
226 //       // as the function signature (minus one, which is the return type).
227 //       for e in &self.equations {
228 //           match e.pattern_length()
229 //               .cmp(&self.arity()) {
230 //               // the equation's pattern is shorter than the function's arity
231 //               // eventually, we'll autocurry this, but for now, we error.
232 //               // TODO: maybe there should be a warning as well?
233 //               Ordering::Less => errs.push(Positional {
234 //                   pos: e.position.clone()
235 //                   , value: format!( "[error] equation had fewer bindings \
236 //                                   than function arity\n \
237 //                                   [error] auto-carrying is not currently \
238 //                                   implemented.\n \
239 //                                   signature: {} \n function: {} \n"
240 //                                   , self.sig
241 //                                   , (*e).to_sexpr(0)
242 //                                   )
243 //               })
244 //               // the equation's pattern is longer than the function's arity
245 //               // this is super wrong and always an error.
246 //               , Ordering::Greater => errs.push(Positional {
247 //                   pos: e.position.clone()
248 //                   , value: format!( "[error] equation bound too many arguments\n \
249 //                                   signature: {} \n function: {} \n"
250 //                                   , self.sig
251 //                                   , (*e).to_sexpr(0)
252 //                                   )
253 //               })
254 //               , _ => {}
255 //           }
256 //       }
257 //       // TODO: this could be made way more idiomatic...
258 //       try_vec!(errs);
259 //       unimplemented!()
260 //   }
261 //
262 // }
263 //
264 //
265 //
266 // // impl TranslateType for types::Type {
267 // //     fn translate_type(&self, context: LLVMContext) -> TypeResult {
268 // //         match *self {
269 // //             types::Type::Ref(ref r) => r.translate_type(context)
270 // //             , types::Type::Prim(ref p) => p.translate_type(context)
271 // //             , _ => unimplemented!() // TODO: figure this out
272 // //         }
273 // //     }

```



```

274 // // }
275 //
276 // // impl TranslateType for Reference {
277 // //     fn translate_type(&self, context: LLVMContext) -> TypeResult {
278 // //         unimplemented!() // TODO: figure this out
279 // //     }
280 // // }
281 //
282 // // impl TranslateType for Primitive {
283 // //     fn translate_type(&self, context: LLVMContext) -> TypeResult {
284 // //         // Ok(match *self {
285 // //             //     Primitive::IntSize => context.int_type(word_size())
286 // //             //     , Primitive::UIntSize => context.int_type(word_size())
287 // //             //     , Primitive::Int(bits) => context.int_type(bits as usize)
288 // //             //     , Primitive::UInt(bits) => context.int_type(bits as usize)
289 // //             //     , Primitive::Float => context.float_type()
290 // //             //     , Primitive::Double => context.double_type()
291 // //             //     , Primitive::Byte => context.byte_type()
292 // //             //     , _ => unimplemented!() // TODO: figure this out
293 // //         })
294 // //         unimplemented!()
295 // //     }
296 // // }

```

B.2 Mnemosyne Parser Crate

lib.rs

```

1 //
2 // Mnemosyne: a functional systems programming language.
3 // (c) 2015 Hawk Weisman
4 //
5 // Mnemosyne is released under the MIT License. Please refer to
6 // the LICENSE file at the top-level directory of this distribution
7 // or at https://github.com/hawkw/mnemosyne/.
8 //
9
10 extern crate combine;
11 extern crate combine_language;
12 extern crate mnemosyne as core;
13
14 use combine::*;
15 use combine_language::{ LanguageEnv
16                       , LanguageDef
17                       , Identifier
18                       };
19 use combine::primitives::{ Stream
20                          , Positioner
21                          , SourcePosition
22                          };
23 use core::chars;

```

```

24 use core::semantic::*;
25 use core::semantic::annotations::{ Annotated
26                                   , UnscopedState
27                                   , Unscoped
28                                   };
29 use core::semantic::types::*;
30 use core::semantic::ast::*;
31 use core::position::*;
32
33 use std::rc::Rc;
34
35 type ParseFn<'a, I, T> = fn (&MnEnv<'a, I>, State<I>) -> ParseResult<T, I>;
36
37 type U = UnscopedState;
38
39 mod tests;
40
41 /// Wraps a parsing function with a language definition environment.
42 ///
43 /// TODO: this could probably push identifiers to the symbol table here?
44 #[derive(Copy)]
45 struct MnParser<'a: 'b, 'b, I, T>
46 where I: Stream<Item=char>
47     , I::Range: 'b
48     , I: 'b
49     , I: 'a
50     , T: 'a {
51     env: &'b MnEnv<'a, I>
52     , parser: ParseFn<'a, I, T>
53 }
54
55 impl<'a, 'b, I, T> Clone for MnParser<'a, 'b, I, T>
56 where I: Stream<Item=char>
57     , I::Range: 'b
58     , I: 'b
59     , T: 'a
60     , 'a: 'b {
61
62     fn clone(&self) -> Self {
63         MnParser { env: self.env , parser: self.parser }
64     }
65 }
66
67 impl<'a, 'b, I, T> Parser for MnParser<'a, 'b, I, T>
68 where I: Stream<Item=char>
69     , I::Range: 'b
70     , I: 'b
71     , T: 'a
72     , 'a: 'b {
73
74     type Input = I;
75     type Output = T;

```

```

76
77     fn parse_state(&mut self, input: State<I>) -> ParseResult<T, I> {
78         (self.parser)(self.env, input)
79     }
80
81 }
82
83 struct MnEnv<'a, I>
84 where I: Stream<Item = char>
85     , I::Item: Positioner<Position = SourcePosition>
86     , I: 'a {
87     env: LanguageEnv<'a, I>
88 }
89
90 impl <'a, I> std::ops::Deref for MnEnv<'a, I>
91 where I: Stream<Item=char>
92     , I: 'a {
93     type Target = LanguageEnv<'a, I>;
94     fn deref(&self) -> &LanguageEnv<'a, I> { &self.env }
95 }
96
97 impl<'a, 'b, I> MnEnv<'a, I>
98 where I: Stream<Item=char>
99     , I::Item: Positioner<Position = SourcePosition>
100     , I::Range: 'b {
101
102     /// Wrap a function into a MnParser with this environment
103     fn parser<T>(&'b self, parser: ParseFn<'a, I, T>)
104         -> MnParser<'a, 'b, I, T> {
105         MnParser { env: self, parser: parser }
106     }
107
108     #[allow(dead_code)]
109     fn parse_def(&self, input: State<I>) -> ParseResult<Form<'a, U>, I> {
110         let function_form
111             = self.name()
112               .and(self.function())
113               .map(|(name, fun)| DefForm::Function { name: name
114                                                         , fun: fun });
115         let top_level
116             = self.name()
117               .and(self.type_name())
118               .and(self.expr())
119               .map(|((name, ty), body)|
120                 DefForm::TopLevel { name: name
121                                     , annot: ty
122                                     , value: Rc::new(body) });
123
124         self.reserved("def").or(self.reserved("define"))
125             .with(function_form.or(top_level))
126             .map(Form::Define)
127             .parse_state(input)

```

```

128 }
129
130 #[allow(dead_code)]
131 fn parse_if(&self, input: State<I>) -> ParseResult<Form<'a, U>, I> {
132     self.reserved("if")
133         .with(self.expr())
134         .and(self.expr())
135         .and(optional(self.expr()))
136         .map(|((cond, if_clause), else_clause)|
137             Form::If { condition: Rc::new(cond)
138                       , if_clause: Rc::new(if_clause)
139                       , else_clause: else_clause.map(Rc::new)
140                     })
141         .parse_state(input)
142 }
143
144 #[allow(dead_code)]
145 fn parse_lambda(&self, input: State<I>) -> ParseResult<Form<'a, U>, I> {
146     self.reserved("lambda")
147         .or(self.reserved(chars::LAMBDA))
148         .with(self.function())
149         .map(Form::Lambda)
150         .parse_state(input)
151 }
152
153 #[allow(dead_code)]
154 fn parse_function(&self, input: State<I>) -> ParseResult<Function<'a, U>, I> {
155     let fn_kwd = choice([ self.reserved("fn")
156                          , self.reserved("lambda")
157                          , self.reserved(chars::LAMBDA)
158                        ]);
159
160     self.parens(fn_kwd.with(
161         self.signature()
162         .and(many1(self.equation()))
163         .map(|(sig, eqs)| Function { sig: sig
164                                     , equations: eqs
165                                   })
166     ))
167     .parse_state(input)
168 }
169
170 #[allow(dead_code)]
171 fn parse_primitive_ty(&self, input: State<I>) -> ParseResult<Type, I> {
172     choice([ self.reserved("int")
173             .with(value(Primitive::IntSize))
174             , self.reserved("uint")
175             .with(value(Primitive::IntSize))
176             , self.reserved("float")
177             .with(value(Primitive::Float))
178             , self.reserved("double")
179             .with(value(Primitive::Double))

```

```

180         , self.reserved("bool")
181           .with(value(Primitive::Bool))
182         , self.reserved("i8")
183           .with(value(Primitive::Int(Int::Int8)))
184         , self.reserved("i16")
185           .with(value(Primitive::Int(Int::Int16)))
186         , self.reserved("i32")
187           .with(value(Primitive::Int(Int::Int32)))
188         , self.reserved("i64")
189           .with(value(Primitive::Int(Int::Int64)))
190     , self.reserved("u8")
191       .with(value(Primitive::Uint(Int::Int8)))
192     , self.reserved("u16")
193       .with(value(Primitive::Uint(Int::Int16)))
194     , self.reserved("u32")
195       .with(value(Primitive::Uint(Int::Int32)))
196     , self.reserved("u64")
197       .with(value(Primitive::Uint(Int::Int64)))
198   ])
199   .map(|primitive| Type::Prim(primitive))
200   .parse_state(input)
201 }
202
203 pub fn raw_ptr_ty(&self, input: State<I>) -> ParseResult<Type, I> {
204     char('`').with(self.type_name())
205       .map(|t| Type::Ref(Reference::Raw(Rc::new(t))))
206       .parse_state(input)
207 }
208
209 pub fn unique_ptr_ty(&self, input: State<I>) -> ParseResult<Type, I> {
210     char('@').with(self.type_name())
211       .map(|t| Type::Ref(Reference::Unique(Rc::new(t))))
212       .parse_state(input)
213 }
214
215 pub fn borrow_ptr_ty(&self, input: State<I>) -> ParseResult<Type, I> {
216     char('&').with(self.type_name())
217       .map(|t| Type::Ref(Reference::Borrowed(Rc::new(t))))
218       .parse_state(input)
219 }
220
221 fn parse_type(&self, input: State<I>) -> ParseResult<Type, I> {
222     choice([ self.parser(MnEnv::parse_primitive_ty)
223             , self.parser(MnEnv::raw_ptr_ty)
224             , self.parser(MnEnv::unique_ptr_ty)
225             , self.parser(MnEnv::borrow_ptr_ty)
226             ])
227       .parse_state(input)
228 }
229
230 fn parse_name_deref(&self, input: State<I>) -> ParseResult<NameRef, I> {
231     char('*').with(self.name())

```

```

232         .parse_state(input)
233     }
234
235     fn parse_name_unique(&self, input: State<I>) -> ParseResult<NameRef, I> {
236         char('@').with(self.name())
237         .map(NameRef::Unique)
238         .parse_state(input)
239     }
240
241     fn parse_name_borrow(&self, input: State<I>)-> ParseResult<NameRef, I> {
242         char('&').with(self.name())
243         .map(NameRef::Borrowed)
244         .parse_state(input)
245     }
246
247     fn parse_owned_name(&self, input: State<I>) -> ParseResult<NameRef, I> {
248         self.name()
249         .map(NameRef::Owned)
250         .parse_state(input)
251     }
252
253     fn parse_name_ref(&self, input: State<I>)
254         -> ParseResult<Form<'a, U>, I> {
255         choice([ self.parser(MnEnv::parse_name_deref)
256             , self.parser(MnEnv::parse_name_unique)
257             , self.parser(MnEnv::parse_name_borrow)
258             , self.parser(MnEnv::parse_owned_name)
259             ])
260         .map(Form::NameRef)
261         .parse_state(input)
262     }
263
264     // fn parse_typeclass_arrow(&self, input: State<I>) -> ParseResult<Estr, I> {
265     //     self.reserved_op("=>")
266     //         .or(self.reserved_op(FAT_ARROW))
267     //         .parse_state(input)
268     // }
269
270     // fn parse_arrow(&self, input: State<I>) -> ParseResult<Estr, I> {
271     //     self.reserved_op(">")
272     //         .or(self.reserved_op(ARROW))
273     //         .parse_state(input)
274     // }
275
276     fn parse_prefix_constraint(&self, input: State<I>)
277         -> ParseResult<Constraint, I> {
278         self.parens(self.reserved_op("=>")
279             .or(self.reserved_op(chars::FAT_ARROW))
280             .with(self.name())
281             .and(many1(self.name()))) )
282         .map(|(c, gs)| Constraint { typeclass: c
283             , generics: gs })

```

```

284         .parse_state(input)
285     }
286
287     fn parse_infix_constraint(&self, input: State<I>)
288         -> ParseResult<Constraint, I> {
289         self.braces(self.name()
290             .skip(self.reserved_op("=>")
291                 .or(self.reserved_op(chars::FAT_ARROW)))
292             .and(many1(self.name()))) )
293         .map(|(c, gs)| Constraint { typeclass: c
294             , generics: gs })
295         .parse_state(input)
296     }
297
298     fn parse_constraint(&self, input: State<I>)
299         -> ParseResult<Constraint, I> {
300     self.parser(MnEnv::parse_prefix_constraint)
301     .or(self.parser(MnEnv::parse_infix_constraint))
302     .parse_state(input)
303     }
304
305     pub fn constraint(&'b self) -> MnParser<'a, 'b, I, Constraint> {
306     self.parser(MnEnv::parse_constraint)
307     }
308
309     fn parse_prefix_sig(&self, input: State<I>) -> ParseResult<Signature, I> {
310     self.parens(self.reserved_op(">")
311         .or(self.reserved_op(chars::ARROW))
312         .with(optional(many1(self.constraint()))))
313         .and(many1(self.type_name())) )
314         .map(|(cs, glob)| Signature { constraints: cs
315             , typechain: glob })
316         .parse_state(input)
317     }
318
319     fn parse_infix_sig(&self, input: State<I>) -> ParseResult<Signature, I> {
320     self.braces(optional(many1(self.constraint())))
321         .and(sep_by1::< Vec<Type>
322             , _, _>( self.lex(self.type_name())
323                 , self.reserved_op(">")
324                 .or(self.reserved_op(
325                     chars::ARROW)
326                 )
327             )))
328         .map(|(cs, glob)| Signature { constraints: cs
329             , typechain: glob })
330         .parse_state(input)
331     }
332
333     fn parse_signature(&self, input: State<I>) -> ParseResult<Signature, I> {
334
335         // let prefix =

```

```

336      //      self.parens(self.reserved_op(">"))
337      //      .or(self.reserved_op(ARROW))
338      //      .with(optional(many1(self.constraint()))))
339      //      .and(many1(self.type_name())) )
340      //      .map(|(cs, glob)| Signature { constraints: cs
341      //      , typechain: glob });
342      //
343      // let infix =
344      //      self.braces(optional(many1(self.constraint()))
345      //      .and(sep_by1::< Vec<Type>
346      //      , _, _>( self.lex(self.type_name())
347      //      , self.reserved_op(">")
348      //      .or(self.reserved_op(ARROW))
349      //      )))
350      //      .map(|(cs, glob)| Signature { constraints: cs
351      //      , typechain: glob });
352      // prefix.or(infix)
353      // .parse_state(input)
354      self.parser(MnEnv::parse_prefix_sig)
355      .or(self.parser(MnEnv::parse_infix_sig))
356      .parse_state(input)
357  }
358
359  pub fn signature(&'b self) -> MnParser<'a, 'b, I, Signature> {
360      self.parser(MnEnv::parse_signature)
361  }
362
363  fn parse_binding(&self, input: State<I>)
364      -> ParseResult<Unscoped<'a, Binding<'a, U>>, I> {
365      let pos = input.position.clone();
366      self.parser(MnEnv::parse_name)
367      .and(self.type_name())
368      .and(self.expr())
369      .map(|((name, typ), value)|
370          Annotated::new( Binding { name: name
371          , typ: typ
372          , value: Rc::new(value)
373          }
374          , Position::from(pos)
375          ))
376      .parse_state(input)
377  }
378
379  #[allow(dead_code)]
380  fn parse_logical(&self, input: State<I>)
381      -> ParseResult<Logical<'a, U>, I> {
382      let and = self.reserved("and")
383      .with(self.expr())
384      .and(self.expr())
385      .map(|(a, b)| Logical::And { a: Rc::new(a)
386      , b: Rc::new(b)
387      });

```



```

388
389         let or = self.reserved("or")
390             .with(self.expr())
391             .and(self.expr())
392             .map(|(a, b)| Logical::And { a: Rc::new(a)
393                                     , b: Rc::new(b)
394                                     });
395
396         and.or(or)
397             .parse_state(input)
398     }
399
400     pub fn int_const(&'b self) -> MnParser<'a, 'b, I, Literal> {
401         self.parser(MnEnv::parse_int_const)
402     }
403
404     #[allow(dead_code)]
405     fn parse_int_const(&self, input: State<I>) -> ParseResult<Literal, I> {
406         self.integer()
407             .map(Literal::IntConst)
408             .parse_state(input)
409     }
410
411     fn parse_let(&self, input: State<I>) -> ParseResult<Form<'a, U>, I> {
412
413         let binding_form =
414             self.reserved("let")
415                 .with(self.parens(many(self.parens(self.binding()))))
416                 .and(many(self.expr()))
417                 .map(|(bindings, body)| LetForm::Let { bindings: bindings
418                                                         , body: body });
419
420         choice([ binding_form ])
421             .map(Form::Let)
422             .parse_state(input)
423     }
424
425     fn parse_name (&self, input: State<I>) -> ParseResult<Ident, I> {
426         let position = input.position.clone();
427         self.env.identifier::<'b>()
428             .map(|name| Positional { pos: Position::from(position)
429                                     , value: name })
430             .parse_state(input)
431     }
432
433     fn parse_call(&self, input: State<I>) -> ParseResult<Form<'a, U>, I> {
434         self.name()
435             .and(many(self.expr()))
436             .map(|(name, args)| Form::Call { fun: name, body: args })
437             .parse_state(input)
438     }
439

```

```

440 fn parse_expr(&self, input: State<I>) -> ParseResult<Expr<'a, U>, I> {
441     let pos = Position::from(input.position.clone());
442     self.env.parens(choice([ try(self.call())
443                             , try(self.def())
444                             , try(self.if_form())
445                             , try(self.lambda())
446                             , try(self.let_form())
447                             ]))
448     .or(try(self.int_const()
449             .map(Form::Lit)))
450     .or(try(self.name_ref()))
451     .map(|f| Annotated::new(f, pos) )
452     .parse_state(input)
453 }
454
455 fn parse_pattern(&self, input: State<I>) -> ParseResult<Pattern, I> {
456     let pat_elem =
457         self.name().map(PatElement::Name)
458         .or(self.int_const().map(PatElement::Lit));
459
460     self.parens(many(pat_elem))
461     .parse_state(input)
462 }
463
464 pub fn pattern(&'b self) -> MnParser<'a, 'b, I, Pattern> {
465     self.parser(MnEnv::parse_pattern)
466 }
467
468 fn parse_equation(&self, input: State<I>)
469     -> ParseResult< Annotated< 'a
470                     , Equation< 'a, U>
471                     , I> {
472     let pos = Position::from(input.position.clone());
473     self.parens(self.pattern()
474                 .and(many(self.expr())))
475     .map(|(pat, body)| Annotated::new( Equation { pattern: pat
476                                             , body: body }
477                                     , pos ))
478     .parse_state(input)
479 }
480
481
482 pub fn equation(&'b self) -> MnParser< 'a, 'b, I
483                             , Annotated< 'a
484                             , Equation<'a, U>
485                             , U>
486                             > {
487     self.parser(MnEnv::parse_equation)
488 }
489
490 pub fn expr(&'b self) -> MnParser<'a, 'b, I, Expr<'a, U>> {
491     self.parser(MnEnv::parse_expr)

```

```

492     }
493
494     pub fn def(&'b self) -> MnParser<'a, 'b, I, Form<'a, U>> {
495         self.parser(MnEnv::parse_def)
496     }
497
498     pub fn if_form(&'b self) -> MnParser<'a, 'b, I, Form<'a, U>> {
499         self.parser(MnEnv::parse_if)
500     }
501
502     pub fn let_form(&'b self) -> MnParser<'a, 'b, I, Form<'a, U>> {
503         self.parser(MnEnv::parse_let)
504     }
505
506     pub fn lambda(&'b self) -> MnParser<'a, 'b, I, Form<'a, U>> {
507         self.parser(MnEnv::parse_lambda)
508     }
509
510     pub fn call(&'b self) -> MnParser<'a, 'b, I, Form<'a, U>> {
511         self.parser(MnEnv::parse_call)
512     }
513
514     pub fn name(&'b self) -> MnParser<'a, 'b, I, Ident> {
515         self.parser(MnEnv::parse_name)
516     }
517
518     pub fn name_ref(&'b self) -> MnParser<'a, 'b, I, Form<'a, U>> {
519         self.parser(MnEnv::parse_name_ref)
520     }
521
522
523     pub fn binding(&'b self)
524         -> MnParser<'a, 'b, I, Unscoped<'a, Binding<'a, U>>> {
525         self.parser(MnEnv::parse_binding)
526     }
527
528     pub fn type_name(&'b self) -> MnParser<'a, 'b, I, types::Type> {
529         self.parser(MnEnv::parse_type)
530     }
531
532     pub fn function(&'b self) -> MnParser<'a, 'b, I, Function<'a, U>> {
533         self.parser(MnEnv::parse_function)
534     }
535
536 }
537 pub fn parse_module<'a>(code: &'a str)
538     -> Result< Vec<Expr<'a, UnscopedState>>
539         , ParseError<&'a str>>
540 {
541     let env = LanguageEnv::new(LanguageDef {
542         ident: Identifier {
543             start: letter().or(satisfy(move |c| chars::ALPHA_EXT.contains(c)))

```

```

544     , rest: alpha_num().or(satisfy(move |c| chars::ALPHA_EXT.contains(c)))
545     , reserved: [ // a number of these reserved words have no meaning yet
546                   "and"                , "begin"
547                   , "case"              , "cond"          , "class"
548                   , "data"
549                   , "define"            , "defn"          , "def"
550                   , "delay"             , "fn"
551                   , "do"                , "else"
552                   , "if"                , "lambda"       , chars::LAMBDA
553                   , "let"               , "let*"        , "letrec"
554                   , "or"
555                   , "quasiquote"        , "quote"        , "unquote"
556                   , "set!"              , "unquote-splicing"
557                   , "struct"            , "union"
558                   , "i8"                , "u8"
559                   , "i16"               , "u16"
560                   , "i32"               , "u32"          , "f32"
561                   , "i64"               , "u64"          , "f64"
562                   , "int"               , "uint"         , "float"
563                   , "bool"              , "double"
564                   , "ref"               , "move"         , "borrow"
565                   , "trait"             , "typeclass"
566                   , "instance"          , "impl"
567                   ].iter().map(|x| (*x).into())
568                   .collect()
569     }
570     , op: Identifier {
571         start: satisfy(move |c| chars::OPS.contains(c))
572         , rest: satisfy(move |c| chars::OPS.contains(c))
573         , reserved: [ ">=", "->", "\\\"", "|", chars::ARROW, chars::FAT_ARROW]
574           .iter().map(|x| (*x).into()).collect()
575     }
576     , comment_line: string(";").map(|_| ())
577     , comment_start: string("#|").map(|_| ())
578     , comment_end: string("#|").map(|_| ())
579   });
580   let env = MnEnv { env: env };
581
582   env.white_space()
583     .with(many1::<Vec<Expr<'a, U>>, _>(env.expr()))
584     .parse(code)
585     .map(|(e, _)| e)
586 }

```

tests.rs

```

1  use super::parse_module;
2
3  use core::semantic::ast::Node;
4
5  macro_rules! expr_test {
6      ($name:ident, $code:expr) => {

```

```

7      #[test]
8      fn $name() {
9          assert_eq!( parse_module($code)
10                      .unwrap()[0]
11                      .to_sexpr(0)
12                      , $code)
13      }
14  }
15 }
16
17 expr_test!(test_basic_add, "(+ 1 2)");
18 expr_test!(test_basic_sub, "(- 3 4)");
19 expr_test!(test_basic_div, "(/ 5 6)");
20 expr_test!(test_basic_mul, "(* 1 2)");
21 expr_test!(test_nested_arith_1, "(+ 1 (- 2 3))");
22 expr_test!(test_nested_arith_2, "(* (+ 1 2) 3 4)");
23 expr_test!(test_nested_arith_3, "(+ (/ 1 2) (* 3 4))");
24
25 expr_test!(test_call_1, "(my_fn 1 2)");
26 expr_test!(test_call_2, "(my_fn (my_other_fn a_var a_different_var))");
27 expr_test!(test_call_3
28     , "(my_fn (my_other_fn a_var a_different_var) VarWithUppercase Othervar)");
29 expr_test!(test_call_4
30     , "(my_fn (my_other_fn a_var a_different_var) (another_fn a_var))");
31
32 expr_test!(test_call_ptr_1, "(my_fn a *b)");
33 expr_test!(test_call_ptr_2, "(my_fn *a *b)");
34 expr_test!(test_call_ptr_3, "(my_fn &a)");
35 expr_test!(test_call_ptr_4, "(my_fn a &b)");
36 expr_test!(test_call_ptr_5, "(my_fn &a &b)");
37 expr_test!(test_call_ptr_6, "(my_fn @a)");
38 expr_test!(test_call_ptr_7, "(my_fn a @b)");
39 expr_test!(test_call_ptr_8, "(my_fn @a @b)");
40
41 expr_test!(test_defsyntax_1,
42     "(define fac (\u{3bb} (\u{8594} int int)
43     \t((0) 1)
44     \t((n) (fac (- n 1))))\n)");
45
46 #[test]
47 fn test_defsyntax_sugar() {
48     let string =
49 r#"
50 (def fac (fn {int -> int}
51     ((0) 1)
52     ((n) (fac (- n 1)))))"#;
53     assert_eq!( parse_module(string).unwrap()[0]
54                 .to_sexpr(0)
55                 , "(define fac (\u{3bb} (\u{8594} int int)
56     \t((0) 1)
57     \t((n) (fac (- n 1)))))\n" )
58 }

```

B.3 Manganese Application Crate

main.rs

```

1  //
2  // The Manganese Mnemosyne Compilation System
3  // (c) 2015 Hawk Weisman
4  //
5  // Mnemosyne is released under the MIT License. Please refer to
6  // the LICENSE file at the top-level directory of this distribution
7  // or at https://github.com/hawkw/mnemosyne/.
8  //
9  extern crate clap;
10 extern crate mnemosyne;
11 extern crate mnemosyne_parser as parser;
12
13 use clap::{Arg, App, SubCommand};
14
15 use std::error::Error;
16 use std::io::Read;
17 use std::fs::File;
18 use std::path::PathBuf;
19
20 use mnemosyne::ast;
21 use mnemosyne::ast::Node;
22 use mnemosyne::errors::UnwrapICE;
23
24 const VERSION_MAJOR: u32 = 0;
25 const VERSION_MINOR: u32 = 1;
26
27 fn main() {
28     let matches = App::new("Manganese")
29         .version(&format!("v{}.{} for {} ({})"
30             , VERSION_MAJOR
31             , VERSION_MINOR
32             , mnemosyne::mnemosyne_version()
33             , mnemosyne::llvm_version()
34         ))
35         .author("Hawk Weisman <hi@hawkweisman.me>")
36         .about("[Mn] Manganese: The Mnemosyne Compilation System")
37         .args_from_usage(
38             "<INPUT> 'Source code file to compile'
39             -d, --debug 'Display debugging information'")
40         .get_matches();
41
42     let path = matches.value_of("INPUT")
43         .map(PathBuf::from)
44         .unwrap();
45
46     let code = File::open(&path)
47         .map_err(|error| String::from(error.description()))
48         .and_then(|mut file| {
49             let mut s = String::new();

```

```

50         file.read_to_string(&mut s)
51         .map_err(|error| String::from(error.description())) )
52         .map(|_| s)
53     })
54     .unwrap();
55
56     let ast = parser::parse_module(code.as_ref())
57         .unwrap();
58
59     for node in ast { println!("{}", (*node).to_sexpr(0)) }
60 }

```

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>.
- [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>.
- [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 Neamtii. “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 p. 2).
- [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.

- [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>.
- [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>.
- [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>.
- [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>.
- [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 p. 6).
- [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>.
- [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.
- [23] 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>.
- [24] 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).
- [25] Tony Hoare. “Null references: The billion dollar mistake”. In: *Presentation at QCon London* (2009).
- [26] Paul Hudak and Joseph H Fasel. “A gentle introduction to Haskell”. In: *ACM Sigplan Notices* 27.5 (1992), pp. 1–52.
- [27] 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.

- [28] 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 p. 1).
- [29] John Hughes. “Why functional programming matters”. In: *The Computer Journal* 32.2 (1989), pp. 98–107.
- [30] Graham Hutton and Erik Meijer. *Monadic parser combinators*. Tech. rep. NOTTCS-TR-96-4. 1996. URL: <http://eprints.nottingham.ac.uk/237/>.
- [31] “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.
- [32] ISO. *C11 Standard*. ISO/IEC 9899:2011 (cit. on p. 2).
- [33] ISO. *C99 Standard*. ISO/IEC 9899:1999 (cit. on p. 2).
- [34] Simon L Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003 (cit. on p. 1).
- [35] Brian W Kernighan, Dennis M Ritchie, and Per Eklint. *The C programming language*. Vol. 2. Prentice-Hall Englewood Cliffs, 1988 (cit. on p. 2).
- [36] 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>.
- [37] 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>.
- [38] Chris Lattner. “LLVM and Clang: Next generation compiler technology”. In: *The BSD Conference*. 2008, pp. 1–2 (cit. on p. 2).
- [39] 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>.
- [40] Daan Leijen and Erik Meijer. “Parsec: Direct style monadic parser combinators for the real world”. In: (2002).
- [41] 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>.
- [42] Luc Maranget. “Compiling pattern matching to good decision trees”. In: *Proceedings of the 2008 ACM SIGPLAN workshop on ML*. ACM. 2008, pp. 35–46.

- [43] Luc Maranget. “Warnings for pattern matching”. In: *Journal of Functional Programming* 17.03 (2007), pp. 387–421.
- [44] 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>.
- [45] 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>.
- [46] 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 p. 7).
- [47] 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. 7).
- [48] Adriaan Moors, Frank Piessens, and Martin Odersky. *Parser combinators in Scala*. Tech. rep. Katholieke Universiteit Leuven, 2008.
- [49] 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).
- [50] Peter Norvig. “Teach yourself programming in ten years”. In: *URL <http://norvig.com/21-days.html#answers>* (2001).
- [51] Martin Odersky et al. *An overview of the Scala programming language*. Tech. rep. 2004 (cit. on p. 1).
- [52] Martin Odersky et al. *The Scala language specification*. 2004 (cit. on p. 1).
- [53] Alan J. Perlis. “Special Feature: Epigrams on Programming”. In: *SIGPLAN Not.* 17.9 (Sept. 1982), pp. 7–13. ISSN: 0362-1340. DOI: 10.1145/947955.1083808. URL: <http://doi.acm.org/10.1145/947955.1083808> (cit. on p. 5).
- [54] 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 p. 2).
- [55] Eric S Raymond. “How to become a hacker”. In: *Database and Network Journal* 33.2 (2003), pp. 8–9.

- [56] 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. 7).
- [57] 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, 6).
- [58] 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. 7).
- [59] 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>.
- [60] Patrick G Sobalvarro. “A lifetime-based garbage collector for LISP systems on general-purpose computers”. PhD thesis. Massachusetts Institute of Technology, 1988.
- [61] Michael Sperber et al. “Revised⁶ report on the algorithmic language Scheme”. In: *Journal of Functional Programming* 19.S1 (2009), pp. 1–301.
- [62] 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.
- [63] Gerry Sussman, Harold Abelson, and Julie Sussman. *Structure and interpretation of computer programs*. MIT Press, Cambridge, Mass, 1983.
- [64] S Doaitse Swierstra. “Combinator parsers: From toys to tools”. In: *Electronic Notes in Theoretical Computer Science* 41.1 (2001), pp. 38–59.
- [65] 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.
- [66] 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>.

- [67] Luke VanderHart and Stuart Sierra. “Macros and Metaprogramming”. In: *Practical Clojure*. Springer, 2010, pp. 167–178.
- [68] 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.
- [69] David Wheeler and Alan Gloria. *Sweet-expressions (t-expressions)*. Tech. rep. SRFI-110. 2006. URL: <http://srfi.schemers.org/srfi-110/srfi-110.html>.
- [70] 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>.