Technical Report CS??-??

Mnemosyne: A Functional Systems Programming Language

Hawk Weisman

Submitted to the Faculty of
The Department of Computer Science

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

Allegheny College
2015

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

_____

Hawk Weisman

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

Abstract

While programming languages researchers have produced a number of languages with various qualities that promote effective programming, such languages often may not be used for low-level systems programming. The automation of memory management through garbage collection often limits the use of many languages for systems programming tasks.

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

List of Figures

List of listings

Chapter 1

Introduction

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

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

[2] Or frightening incompetence

Chapter 2

Background and Rationale

Chapter 3

Design Considerations

3.1 Design Goals

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

Expressiveness

3.2 Characteristics

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

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

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 cadegory 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.

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.

3.3 Experiments

3.4 Threats to Validity

Chapter 4

Implementation

4.1 Parsing and Syntactic Analysis

4.2 Semantic Analysis

4.3 Code Generation

Chapter 5

Evaluation

Another possible chapter title: Experimental Results

Chapter 6

Discussion and Future Work

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

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.0.1 Syntactic Notation

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

⟨*symbol*⟩ indicates a non-terminal symbol

‘`symbol`’ indicates a terminal symbol

⟨*symbol*⟩* indicates zero or more repetitions of ⟨*symbol*⟩

⟨*symbol*⟩+ indicates one or more repetitions of ⟨*symbol*⟩.

ε indicates the empty string

The following special symbols refer to specific Unicode characters:

⟨*lambda*⟩ Greek capital letter Lambda (U+03BB)

⟨*arrow*⟩ Rightwards arrow (U+2192)

⟨*double arrow*⟩ Rightwards double arrow (U+21D2)

⟨*tab*⟩ Character tabulation (U+0009)

⟨*linefeed*⟩ Linefeed (U+000A)

⟨*return*⟩ Carriage return (U+000D)

⟨*space*⟩ Space (U+0020)

Finally, the symbol ⟨*any*⟩ refers to any character.

A.1 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.4) and *expressions* (Appendix A.3). 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.2 Lexical Syntax

This section describes the lexical structure of Mnemosyne programs.

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

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

⟨*program*⟩ → ⟨*token*⟩+

⟨*token*⟩ → ⟨*lexeme*⟩ | ⟨*atmosphere*⟩

⟨*lexeme*⟩ → ⟨*identifier*⟩ | ⟨*operator*⟩ | ⟨*keyword*⟩ | ⟨*literal*⟩
| ⟨*sigil*⟩ | ⟨*delimiter*⟩

⟨*sigil*⟩ → '@' | '&' | '*' | '$' | '?'

⟨*delimiter*⟩ → '(' | ')' | '{' | '}'

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

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

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

⟨*letter*⟩ → 'a' | 'b' | 'c' | ... | 'z'
| 'A' | 'B' | 'C' | ... | 'Z'

⟨*number*⟩ → '0' | '1' | ... | '9'

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

⟨*special subsequent*⟩ → ⟨*special initial*⟩ | ''' | '_'

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

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

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

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

⟨*comment*⟩ → ';' ⟨*any*⟩* ⟨*line ending*⟩
| '#|' ⟨*any*⟩* '|#'

A.3 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.

⟨*expr*⟩ → ⟨*s-expr*⟩ | ⟨*i-expr*⟩ | ⟨*c-expr*⟩ | ⟨*n-expr*⟩
| ⟨*deref expr*⟩ | ⟨*unwrap expr*⟩ | ⟨*pointer expr*⟩
| ⟨*literal*⟩

⟨*s-expr*⟩ → '(' ⟨*operator*⟩ ⟨*expr*⟩* ')'

⟨*c-expr*⟩ → '{' ⟨*expr*⟩ ⟨*operator*⟩ ⟨*c-expr body*⟩+ '}'

⟨*c-expr body*⟩ → ⟨*expr*⟩ ⟨*operator*⟩
| ⟨*expr*⟩

⟨*n-expr*⟩ → ⟨*access*⟩ '(' ⟨*expr*⟩* ')'
| ⟨*access*⟩ '.' ⟨*identifier*⟩
| ⟨*access*⟩ '.' ⟨*n-expr*⟩

⟨*access*⟩ → ⟨*deref-expr*⟩ | ⟨*identifier*⟩

⟨*deref expr*⟩ → '(' '$' ⟨*expr*⟩ ')'
| '$' ⟨*expr*⟩

⟨*unwrap expr*⟩ → '(' '?' ⟨*expr*⟩ ⟨*expr*⟩')'
| '(' '?' ⟨*expr*⟩ ')'
| '?'⟨*expr*⟩

⟨*pointer expr*⟩ → '(' ⟨*pointer sigil*⟩ ⟨*expr*⟩ ')'
| ⟨*pointer sigil*⟩ ⟨*expr*⟩

⟨*pointer sigil*⟩ → '&' | '@' | '*'

A.3.1 Program Structure

⟨*program*⟩ → ⟨*module*⟩+

⟨*module*⟩ → ⟨*module-def*⟩ ⟨*definition*⟩*

⟨*module-def*⟩ → '(' 'mod' ⟨*identifier*⟩ ⟨*exports-clause*⟩ ')'

⟨*exports-clause*⟩ → '(' 'exports' ⟨*identifier*⟩+ ')'
| ε

A.4 Definitions

A.4.1 Attributes

⟨*function-attrs*⟩ → '#(' ⟨*function-attr*⟩* ')'

⟨*function-attr*⟩ → ⟨*any-attr*⟩
| ‘`cold`’
| ⟨*inline*⟩

⟨*inline*⟩ → ‘`inline`’
| ‘`inline-always`’

⟨*data-attr*⟩ → ⟨*any-attr*⟩
| ‘`(`’ ‘`as`’ ⟨*as-attr*⟩+ ‘`)`’
| ‘`as`’ ‘`(`’ ⟨*as-attr*⟩+ ‘`)`’

⟨*as-attr*⟩ → ‘`C`’ | ‘`packed`’
| ‘`u8`’ | ‘`u16`’ | ‘`u32`’ | ‘`u64`’

Appendix B

Manganese Source Code

All program code should be fully commented. Authorship of all parts of the code should be clearly specified.

B.1 Mnemosyne Core Crate

lib.rs

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

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

extern crate rustc;
extern crate libc;
extern crate combine;
extern crate iron_llvm;
```

```
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

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

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

errors.rs

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

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

/// Mnemosyne error handling

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

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

```
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 Mnemosyne 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 mnemosyne::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 Mnemosyne 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 Mnemosyne
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");
```

```
    }

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

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

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

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

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

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

forktable.rs

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

```
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 implemention 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/scal
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 mnemosyne::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 mnemosyne::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 mnemosyne::forktable::ForkTable;
128     /// let mut table: ForkTable<isize,&str> = ForkTable::new();
129     /// assert_eq!(table.get_mut(&1), None);
130     /// table.insert(1isize, "One");
131     /// assert_eq!(table.get_mut(&1), Some(&mut "One"));
132     /// assert_eq!(table.get_mut(&2), None);
133     /// ```
134     /// ```ignore
135     /// # use mnemosyne::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 mnemosyne::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 mnemosyne::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 mnemosyne::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 mnemosyne::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 mnemosyne::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 mnemosyne::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 mnemosyne::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 mnemosyne;
49      /// # use combine::primitives::SourcePosition;
50      /// # use mnemosyne::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 mnemosyne::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
```

```
                        }

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

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


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

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

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


/// This is literally just waving my hands for the compiler.
///
/// Hopefully it understands what I mean.
impl<T> Eq for Positional<T>
where T: Eq
    , T: PartialEq
    {}

impl<T> Deref for Positional<T> {
```

```
    type Target = T;
    fn deref(&self) -> &T {
        &self.value
    }
}

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

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

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

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

compile/mod.rs

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

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

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

use iron_llvm::core;
use iron_llvm::core::types::{ Type
```

```
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 occured 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/jauhien/iron-kaleidoscope)
93  /// tutorial, and from [`librustc_trans`](https://github.com/rust-lang/rust/blob/master/src/libr
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-currying is not currently \
238 //                                      implemented.\n \
239 //                                      signature: {}\nfunction: {}\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: {}\nfunction: {}\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
```

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

B.2 Mnemosyne Parser Crate

lib.rs

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

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

use combine::*;
use combine_language::{ LanguageEnv
                      , LanguageDef
                      , Identifier
                      };
use combine::primitives::{ Stream
                         , Positioner
                         , SourcePosition
                         };
use core::chars;
use core::semantic::*;
use core::semantic::annotations::{ Annotated
                                 , UnscopedState
                                 , Unscoped
                                 };
```

```
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
```

```
}

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

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

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

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

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

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

    #[allow(dead_code)]
    fn parse_if(&self, input: State<I>) -> ParseResult<Form<'a, U>, I> {
        self.reserved("if")
```

```
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")
```

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

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

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

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

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

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

```
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<&str, 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<&str, 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)
```

```
                                                , b: Rc::new(b)
                                                });

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

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

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

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

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

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

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

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

    fn parse_expr(&self, input: State<I>) -> ParseResult<Expr<'a, U>, I> {
        let pos = Position::from(input.position.clone());
        self.env.parens(choice([ try(self.call())
                               , try(self.def())
                               , try(self.if_form())
```

```
                                , try(self.lambda())
                                , try(self.let_form())
                                ]))
            .or(try(self.int_const()
                        .map(Form::Lit)))
            .or(try(self.name_ref()))
            .map(|f| Annotated::new(f, pos) )
            .parse_state(input)
    }

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

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

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

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

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

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

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

```
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"
```

```
                        , "define"              , "defn"        , "def"
                        , "delay"               , "fn"
                        , "do"                  , "else"
                        , "if"                  , "lambda"      , chars::LAMBDA
                        , "let"                 , "let*"        , "letrec"
                        , "or"
                        , "quasiquote"          , "quote"       , "unquote"
                        , "set!"                , "unquote-splicing"
                        , "struct"              , "union"
                        , "i8"                  , "u8"
                        , "i16"                 , "u16"
                        , "i32"                 , "u32"         , "f32"
                        , "i64"                 , "u64"         , "f64"
                        , "int"                 , "uint"        , "float"
                        , "bool"                                , "double"
                        , "ref"                 , "move"        , "borrow"
                        , "trait"               , "typeclass"
                        , "instance"            , "impl"
                        ].iter().map(|x| (*x).into())
                         .collect()
        }
      , op: Identifier {
            start: satisfy(move |c| chars::OPS.contains(c))
          , rest:  satisfy(move |c| chars::OPS.contains(c))
          , reserved: [ "=>", "->", "\\", "|", chars::ARROW, chars::FAT_ARROW]
                .iter().map(|x| (*x).into()).collect()
        }
      , comment_line: string(";").map(|_| ())
      , comment_start: string("#|").map(|_| ())
      , comment_end: string("|#").map(|_| ())
    });
    let env = MnEnv { env: env };

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

tests.rs

```
use super::parse_module;

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

macro_rules! expr_test {
    ($name:ident, $code:expr) => {
        #[test]
        fn $name() {
            assert_eq!( parse_module($code)
                            .unwrap()[0]
                            .to_sexpr(0)
```

```
                        , $code)
        }
    }
}

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

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

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

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

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

B.3 Manganese Application Crate

main.rs

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

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

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

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

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

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

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

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

```
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

[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.

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

[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] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Jeroen Fokker. “Functional parsers”. In: *Advanced Functional Programming*. Springer, 1995, pp. 1–23.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] Tony Hoare. “Null references: The billion dollar mistake”. In: *Presentation at QCon London* (2009).

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

[24] 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.

[25] 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.

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

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

[28] “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.

[29] Simon L Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[30] Brian W Kernighan, Dennis M Ritchie, and Per Ejeklint. *The C programming language*. Vol. 2. Prentice-Hall Englewood Cliffs, 1988.

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

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

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

[38] 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.

[39] 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.

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

[41] 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.

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

[43] Martin Odersky et al. *An overview of the Scala programming language*. Tech. rep. 2004.

[44] Martin Odersky et al. *The Scala language specification*. 2004.

[45] 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.

[46] 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.

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

[48] 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.

[49] 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.

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

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

[52] 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.

[53] Gerry Sussman, Harold Abelson, and Julie Sussman. *Structure and interpretation of computer programs*. MIT Press, Cambridge, Mass, 1983.

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

[55] 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.

[56] 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.

[57] Luke VanderHart and Stuart Sierra. “Macros and Metaprogramming”. In: *Practical Clojure*. Springer, 2010, pp. 167–178.

[58] 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.

[59] 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.