

# Mnemosyne: A Functional Language for Systems Programming

## CMPSC600 Senior Thesis Proposal

Hawk Weisman

Department of Computer Science  
Allegheny College

November 20<sup>th</sup>, 2015

# Proposal

To implement and evaluate a prototype compiler for the Mnemosyne programming language.

- ▶ Mnemosyne is a functional language for systems programming, with compile-time automatic memory management.
- ▶ But what does that mean?

# What is Mnemosyne?

A functional language for systems programming, with compile-time automatic memory management.

# What is Mnemosyne?

A functional language for systems programming, with compile-time automatic memory management.

- ▶ **Functional programming** models computation as the evaluation of functions [12, 26]
  - ▶ It focuses on immutability, purity, and function composition

# What is Mnemosyne?

A functional language for systems programming, with compile-time automatic memory management.

- ▶ **Functional programming** models computation as the evaluation of functions [12, 26]
  - ▶ **It focuses on** immutability, purity, and function composition
  - ▶ **Advantages:** expressiveness [10, 12], modularity [10, 12], safety

# What is Mnemosyne?

A functional language for systems programming, with compile-time automatic memory management.

- ▶ **Mnemosyne is inspired by:**

- ▶ **Lisp**'s syntax and homoiconicity [23, 25].
- ▶ **Haskell and ML**'s type system [9, 11, 14] and pattern matching [11, 14, 16, 18]
- ▶ **Rust**'s memory management [2]

# What is Mnemosyne?

A functional language **for systems programming**, with compile-time automatic memory management.

- ▶ **Systems programming** is the implementation of software that provide services to other software [20, 21].

# What is Mnemosyne?

A functional language **for systems programming**, with compile-time automatic memory management.

- ▶ **Systems programming** is the implementation of software that provide services to other software [20, 21].
- ▶ High quality systems are necessary for high quality applications.



# What is Mnemosyne?

A functional language **for systems programming**, with compile-time automatic memory management.

- ▶ **Systems programming** is the implementation of software that provide services to other software [20, 21].
- ▶ High quality systems are necessary for high quality applications.
- ▶ But there are some significant challenges in this field [2, 21]

# What is Mnemosyne?

A functional language for systems programming, **with compile-time automatic memory management.**

- ▶ Almost all systems programming today is done in C [7, 21]

# What is Mnemosyne?

A functional language for systems programming, **with compile-time automatic memory management.**

- ▶ Almost all systems programming today is done in C [7, 21]
- ▶ **Why?** C manages memory at compile-time

# What is Mnemosyne?

A functional language for systems programming, **with compile-time automatic memory management**.

- ▶ Almost all systems programming today is done in C [7, 21]
- ▶ **Why?** C manages memory at compile-time
  - ▶ Most languages use garbage collection (GC) [1]
  - ▶ GC is unsuitable for most low-level systems [7, 8, 21]
  - ▶ C manages memory manually (`malloc()`/`free()`) [8, 15, 21]

# What is Mnemosyne?

A functional language for systems programming, **with compile-time automatic memory management.**

- ▶ **Manual memory management leads to errors** such as buffer overflows, memory leaks, and null pointer dereferences [7, 21]
- ▶ **What if there was another way?**

# What is Mnemosyne?

A functional language for systems programming, **with compile-time automatic memory management.**

- ▶ Mnemosyne manages memory automatically at compile time
- ▶ **How?**

# What is Mnemosyne?

A functional language for systems programming, **with compile-time automatic memory management.**

- ▶ Mnemosyne manages memory automatically at compile time
- ▶ **How?**
  - ▶ Stack allocation [3, 6, 19]
  - ▶ Lending and ownership analysis [19]
  - ▶ Controlled mutability [19]

# Mnemosyne Syntax

Calculating factorials

```
(def factorial (fn ( -> int int )  
  ((factorial 0) 1)  
  ((factorial n) ( * n (factorial (- n 1)) )  
)))
```



# Mnemosyne Syntax

The same example, using syntactic sugar.

```
def factorial (fn { int -> int }  
  (factorial 0) 1  
  (factorial n) {  
    n * factorial({n - 1})  
  })
```

# Methods

Manganese, the Mnemosyne compiler, is implemented in Rust

- ▶ **Combinator parsing** [4, 5, 13, 24] using `combine` and `combine-language`
- ▶ **Analysis** including type checking and lifetime analysis [19, 22]
- ▶ **Code generation** using `librustc-llvm` [17]

# Methods

Assessing Mnemosyne's correctness

- ▶ **Unit and integration testing** to validate the compiler implementation
- ▶ **Demonstration** by implementing example code, including parts of the prelude
- ▶ **Benchmarking** compiled Mnemosyne binaries

# Questions?

For more information:

- ▶ **Sample Mnemosyne code** if there's time
- ▶ **Complete source code:**  
<https://github.com/hawkw/mnemosyne>

# References |



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.



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.



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.



Nils Anders Danielsson. “Total Parser Combinators”. In: *SIGPLAN Not.* 45.9 (Sept. 2010), pp. 285–296. ISSN: 0362-1340.



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.



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.

# References II



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.



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.



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



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.



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.



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

# References III



Graham Hutton and Erik Meijer. *Monadic parser combinators*. Tech. rep. NOTTCS-TR-96-4. 1996.



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



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



Neelakantan R. Krishnaswami. “Focusing on Pattern Matching”. In: *SIGPLAN Not.* 44.1 (Jan. 2009), pp. 366–378. ISSN: 0362-1340.



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.



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

# References IV



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.



Thomas Narten. “Systems Programming”. In: *Encyclopedia of Computer Science*. Chichester, UK: John Wiley and Sons Ltd., pp. 1739–1741. ISBN: 0-470-86412-5.



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.



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



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



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



# References V



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



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.