M Language Documentation

Aedan Smith

Jul 05, 2020

Contents

CHAPTER 1

The M Tutorial

1.1 Introduction

The fundamental idea behind the M programming language is that simplicity is as important as expressiveness; or, put another way, that a program which cannot be reasoned about is little better than no program at all. Many features of modern programming languages complicate programs unnecessarily, and combining these features increases complexity exponentially. For the programmer, this means time wasted trying to figure out what a program does. For the tooling, this means less support, worse performance, and more bugs.

M attempts to replace these complicated language features with straightforward abstractions, making programs simpler and easier to reason about. While many advanced features are still available in M, they are provided through the composition of abstractions rather than as part of the language, allowing them to be programmatically modified or erased. Looking at the implementation of a feature is as simple as looking at the implementation of a function, and removing it is as simple as inlining said function.

Despite this emphasis on simplicity, M maintains (and even exceeds) the expressiveness of most other programming languages. The ability to define new features in M makes adding missing ones simple, and no tooling needs to be updated to support them. Modifications of features can be used along side one another, and complete paradigm switches can be made local to a single function.

If you just want a quick introduction to the semantics of the language, the reference covers everything you need to know. Otherwise, continue on.

1.1.1 Intended Audience

This tutorial is intended as a simple introduction to M, and while it does not assume any prior knowledge, familiarity the Lambda Calculus, other Lisps, and functional programming in general will be useful. Though many of the topics covered are relatively simple, the ways in which they interact can be difficult to understand if you are not already familiar with them. Every concept will be explained at least briefly, but fully exploring concepts like monads is beyond the scope of this tutorial.

1.2 Getting Started

1.2.1 Setting up the M Compiler

The M compiler does not need any special setup to work; simply clone M's github repository in the directory of your choice, install the dependencies, and run.

```
git clone https://github.com/m-language/m.git
cd m/m-ps
bower install
npm install
pulp build
npm run repl
```

1.2.2 Setting up your Editor

There are M plugins for several popular editors:

- VSCode
- Vim
- Sublime
- IDEA

If you have written a plugin for your favorite editor, please open a pull request to have it added to this list.

1.3 Functions

Functions are the fundamental data type of M. Every type, except for a few primitives such as integers and expressions, is represented as a function, even typically intrinsic types like booleans and tuples. Functions in M are pure, meaning that they cannot have side effects like IO or mutation, and only ever take one argument (multi-argument functions are provided as syntax sugar).

1.3.1 Function Application

Function application is of the form `(<fn> <args...>)`, where `fn` is the function to be applied and `args` is a list of arguments to the function.

```
# true
(not false)

# 4
(add 1 3)
```

Functions are applied eagerly, and their evaluation order is undefined.

1.3.2 Function Abstraction

Function abstraction is of the form `(fn <args> <val>)`, where `args` is a list of argument names and `val` is the value of the function.

```
# The identity function which always returns its argument
(fn x x)

# The constant function which ignores its second argument
(fn [x y] x)

# The increment function
(fn x (add x 1))
```

1.3.3 Closures

Closures are a property of functions where the variables of an outside scope are captured by a function. The values of these variables are stored in the function's closure, and persist for as long as the function persists.

```
# The increment function expressed with closures; the variable x is bound
# to 1, and the function of y stores this value in its closure
((fn x (fn y (add x y)))
  1)
```

1.3.4 Currying

Curried functions are functions which use closures to take their arguments one at a time, allowing for partial application of their arguments. In M, all functions are curried, including internal functions.

```
# The increment function
(add 1)
```

1.3.5 Multi-Argument Functions

Multi-argument functions in M are just syntax sugar for curried functions. Likewise, multi-argument application is just syntax sugar for curried application.

```
# The two-argument application function with syntax sugar
(fn [f a b] (f a b))

# The two-argument application function without syntax sugar
(fn f (fn a (fn b ((f a) b))))
```

1.4 Definitions

While it is possible to introduce local variables with functions, something more powerful is needed to abstract in larger programs. Definitions provide a simple way of providing both local and global variables without worrying about includes or forward declarations.

1.4.1 Definition Expressions

Definition expressions are of the form `(def <name> <val>)`, where `name` is the name of the definition and `val` is is the value of the definition.

```
# The identity function which always returns its argument.
(def id (fn x x))

# The constant function which ignores its second argument.
(def const (fn [x y] x))
```

1.4.2 Ordering

M definitions can be put in any order regardless of their dependencies; the M compiler will order the definitions such that a definition is always put before its first usage.

```
# 3
(def three (inc 2))

# The increment function which adds one to its argument
(def inc (add 1))
```

1.4.3 Applying Definitions

Definitions are expressions which return an environment. When an environment is applied, it evaluates its argument within the environment.

```
# 3
((def inc (add 1))
  (inc 2))

# Equivalent to the above due to currying
(def inc (add 1)
  (inc 2))
```

1.4.4 Block Expressions

Block expressions are expressions of the form `(block exprs)`, where `exprs` is the list of expressions in the block. A block expression combines the environments of all expressions in the block to create a single environment.

```
# Defines three and inc
(block {
  (def three (inc 2))
  (def inc (add 1))
})
```

Semantically, the top level of an M program is a block expression.

1.4.5 Cross File Definitions

M's definitions are designed in such a way that two definitions in two different files are equivalent to two definitions in the same file. In other words, it does not matter where the definitions are stored as long as the M compiler can find them.

By default, the M compiler looks in the current directory for definitions. This is semantically equivalent to combining each file in your project into one file.

1.5 Macros

Macros are the core of M's higher level abstractions; they allow you to use information from the compiler to transform expressions and create new semantics. To do this, M treats its own structure as a primitive type, and allows the definition of functions which operate on it.

1.5.1 Macro Application

Macros are applied just like functions, but instead of operating on the values of their arguments, they operate on the expressions of their arguments.

```
# The identity function defined using the defn macro
(defn (id x) x)
```

1.5.2 Creating Expressions

Quote expressions are of the form `(quote <expr>)`, where `expr` is the expression to evaluate to. Quote allows the creation of static expressions which can be manipulated by other functions.

```
# The expression representing def
(quote def)

# The expression representing (fn x x)
(quote (fn x x))
```

1.5.3 Combining Expressions

Expressions can be combined by applying them to other expressions. The application of two expressions is equivalent to the expression representing their application.

```
# The expression representing (fn x x)
((quote fn) qx qx)
(def qx (quote x))
```

1.5.4 Transforming Expressions

Macros are of the form `(fm <args> <val>)`, where where `args` is a list of argument names and `val` is the value of the function macro. When applied to an expression, a macro quotes that expression and transforms it.

```
# A macro similar to defn
(def deffn
  (fm [name args value]
    ((quote def) name
      ((quote fn) args value))))

# The identity function defined using the deffn macro
(deffn id [x] x)
```

1.5.5 Currying

Internally, macros are not curried, as they are required to return expressions rather than functions. However, they can still be treated like they are curried, and will work as expected.

```
# Defines inc with currying
((def inc) (fn x (add 1 x)))

# Equivalent to the above
((defn (inc x)) (add 1 x))
```

CHAPTER 2

Reference

TODO

CHAPTER 3

Indices and tables

- genindex
- modindex
- search