The M Specification

Aedan Smith

April 5, 2019

Contents

1 Introduction

There has always been a clear separation between practical programming languages and mathematical programming constructs. Even languages such as ML and Scheme, though rooted in mathematics, have unnecessary or impure extensions for convenience and performance. While these extensions improve the language in some ways, they also make it more complex; programs become difficult to reason about for humans and computers alike, and maintaining a compliant, feature-rich software stack becomes impossible.

Simplicity is necessary to reason about a programming language; and when writing simple programs, most programming languages are. It is when programs get more complex, be it for performance or for abstraction, that simplicity is discarded; complex extensions are used, edge cases are abused, and reasoning about a program becomes impossible.

M is not just a simple programming language — mathematical systems like Turing Machines and the λ-Calculus are much simpler than M is. M is a simple programming language which can express the extensions provided by practical programming languages without the additional complexity.

To do this, M exhibits the following properties:

1. All functions are pure.
2. All definitions are unordered.
3. Any two expressions can be made equivalent with a macro.

2 Syntax

```
∀ℂ              : ℂ ⊇ (special + newline + whitespace) ∧ null ∉ ℂ
character       = ℂ
special         = {';', '"', '(', ')'}
comment         = ';'  (character - newline)*  newline
symbol          = (character - special - whitespace)*
                | '"'  character*  '"'
                | '"""'  character*  '"""'
expression      = symbol
                | '('  expression*  ')'
program         = expression*
```

Figure 1: The M grammar in EBNF.

2.1 Characters

M can use any character set which encodes the four special characters, a newline character, and a whitespace character. To allow for simple source termination, the null character is not allowed in M code. For consistency, all M code samples in this specification use the ASCII character set.

2.2 Specials

The four special characters are reserved for use in non-symbol syntax. This is done to ensure that symbols do not include any characters which are meant be used in other syntactic constructs. For example, if the special characters were not reserved, the expression `(symbol)` would be parsed as `("symbol)"` rather than `("symbol")`.

2.3 Comments

Comments in M begin with a semicolon and last until the end of the line. They are ignored and discarded like whitespace and newlines, as they are only intended to be used for explaining code. M does not provide multiline comments, as comments are not meant to be used for documentation.

2.4 Symbols

Symbols in M have two forms, inline and literal. Inline symbols are strings terminated by whitespace or a special symbol, and should be used by default. Literal symbols are strings surrounded by one or two quotes, and should only be used when a symbol is impossible to represent using inline symbols (for example, the symbols `"()"` and `""""`).

2.5 Expressions

Expressions in M have two forms, symbols and lists. Symbol expressions are symbols as defined in section 2.4. Lists expressions are lists of expressions surrounded by matching parentheses.

2.6 Programs

M programs are lists of expressions. Note that there is no terminator specified for M programs, nor a separator for expressions, so M programs are not restricted to a single file or a null-terminated string.

3 Semantics

$$\text{(Symbol)}\ \frac{}{\langle x, \Gamma\rangle \Downarrow \Gamma(x)} \qquad \text{(Macro)}\ \frac{\langle e, \Gamma\rangle \Downarrow v}{\langle \texttt{(macro x e)}, \Gamma\rangle \Downarrow \langle v, \Gamma(x) = (v, \top)\rangle}$$

$$\text{(Function)}\ \frac{}{\texttt{(fn x e)} \Downarrow \lambda x.e} \qquad \text{(Apply-Function)}\ \frac{\langle f, \Gamma\rangle \Downarrow \lambda x.e \quad \langle a, \Gamma\rangle \Downarrow i \quad \langle e[x/i]\rangle \Downarrow v}{\langle \texttt{(f a)}, \Gamma\rangle \Downarrow v}$$

$$\text{(Define)}\ \frac{\langle e, \Gamma\rangle \Downarrow v}{\langle \texttt{(def x e)}, \Gamma\rangle \Downarrow \langle v, \Gamma(x) = (v, \bot)\rangle} \qquad \text{(Apply-Macro)}\ \frac{\Gamma(m) = (f, \top) \quad \langle f(e), \Gamma\rangle \Downarrow v}{\langle \texttt{(m e)}, \Gamma\rangle \Downarrow v}$$

Figure 2: The natural semantics of M.

3.1 Symbol

Symbol expressions are expressions of the form `name`. They always evaluate to the `value` of the def expression that defines `name`. If `name` is not defined, they evaluate to $\bot$ instead.

3.2 Function

Function expressions are expressions of the form `(`**`fn`** `name value)`. They evaluate to a function λ`name.value`. When applied, they perform the substitution `value`[`name`/`argument`].

3.3 Define

Def expressions are expressions of the form `(`**`def`** `name value)`. They state that all symbols `name` evaluate to `value`, and evaluate to the the `value` they define. Multiple def expressions with the same `name` are invalid.

3.4 Macro

Macro expressions are expressions of the form `(`**`macro`** `name value)`. They state that all symbols `name` evaluate to `value`, and evaluate to the value of `name`. When applied, macros evaluate `name` and apply it to the expression encoding of `argument`, then evaluate the result.

3.5 Apply

Apply expressions are expressions of the form `(value argument)`. If `value` is a function, it performs application as described in section 3.2. If `value` is a macro, it performs application as described in section 3.4

4 Encodings

```
;;; Booleans

(def true
  (fn x
    (fn _ x)))

(def false
  (fn _
    (fn x x)))

;;; Products

(def pair
  (fn first second
    (fn value
      (value first second))))

(def first
  (fn pair
    (pair true)))

(def second
  (fn pair
    (pair false)))

;;; Coproducts

(def left
  (fn value
    (fn first _
      (first value))))

(def right
  (fn value
    (fn _ second
      (second value))))

(def left?
  (fn either
    (either
      (fn _ true)
      (fn _ false))))

(def right?
  (fn either
    (either
      (fn _ false)
      (fn _ true))))
```

```
;;; Lists

(def nil (left false))

(def cons
  (fn car
    (fn cdr
      (right (pair car cdr)))))

(def car
  (fn value
    (list (fn x x) left)))

(def cdr
  (fn list
    (list (fn x x) right)))

(def nil? left?)

;;; Natural Numbers

(def 0 (left (fn x x)))
(def 0? left?)
(def inc right)
(def dec
  (fn nat
    (nat left (fn x x))))

;;; Chars

(def nat->char (fn x x))
(def char->nat (fn x x))

;;; Symbols

(def list->symbol (fn x x))
(def symbol->list (fn x x))

;;; Expressions

(def symbol? left?)
(def list? right?)
```

5 Reference Implementation

5.1 Parser

TODO

5.2 Interpreter

TODO

5.3 Compiler

TODO