The M Specification

Aedan Smith

February 28, 2019

Contents

1 Syntax

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

Figure 1: The M grammar in EBNF.

1.1 Characters

M can use any character set which encodes the four special characters, a backslash, a newline character, and a whitespace character. For consistency, all M code samples in this specification use the ASCII character set.

1.2 Specials

The four special characters are reserved for use in other productions. This is done to ensure that symbols such as `this()` do not include any special characters which are meant be used for other syntactic constructs. Note that the backslash is not included in the list of special characters as it is only used inside of a literal symbol.

1.3 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.

1.4 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 symbol `"()"`).

1.5 Expressions

Expressions in M have two forms, symbols and lists. Symbol expressions are identical to symbols defined in section 1.4. Lists expressions are simply lists of expressions surrounded by matching parentheses.

1.6 Programs

M programs are lists of expressions. It is unspecified how these expressions are stored, so they can be in memory, in a single file, in multiple files, on the internet, or any other method which is convenient.

2 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.

2.1 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 $\bot$.

2.2 Function

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

2.3 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.

2.4 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.

2.5 Apply

Apply expressions are expressions of the form `(value argument)`. If `value` is a function, it performs application as described in section 2.2. If `value` is a macro, it performs application as described in section 2.4

3 Encodings

```
;;; Functions

(def id (fn x x))

(def const
  (fn x
    (fn "" x)))

;;; 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
      (const true)
      (const false))))

(def right?
  (fn either
    (either
      (const false)
      (const true))))

;;; Lists

(def nil (left false))

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

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

(def cdr
  (fn list
    (list id right)))

(def nil? left?)
```

```
;;; Natural Numbers

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

;;; Chars

(def nat->char id)
(def char->nat id)

;;; Symbols

(def list->symbol id)
(def symbol->list id)

;;; Expressions

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

4 Reference Implementation

4.1 Parser

()

4.2 Interpreter

()

4.3 Compiler

()