The M Specification

Aedan Smith

February 2, 2019

Contents

1 Syntax

⟨*character*⟩ ::= $\mathbb{C} \supseteq$ ⟨*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 escapable strings surrounded by 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

$$\text{(Context)}\quad \frac{a \Downarrow i}{\langle a, \Gamma\rangle \Downarrow \langle i, \Gamma\rangle} \qquad \frac{\langle a, \Gamma\rangle \Downarrow \lambda x.i \qquad \langle b, \Gamma\rangle \Downarrow j \qquad \langle i[x/j]\rangle \Downarrow v}{\langle \texttt{(a b)}, \Gamma\rangle \Downarrow v}\quad \text{(Apply-Lambda)}$$

$$\text{(Do)}\quad \frac{\langle a, \Gamma\rangle \Downarrow i}{\langle \texttt{(}\textbf{do}\texttt{ a)}, \Gamma\rangle \Downarrow \pi i} \qquad \frac{\langle a, \Gamma\rangle \Downarrow \pi i \qquad \langle b, \Gamma\rangle \Downarrow j \qquad \langle j(i)\rangle \Downarrow \pi v}{\langle \texttt{(a b)}, \Gamma\rangle \Downarrow \pi v}\quad \text{(Apply-Do)}$$

$$\text{(Lambda)}\quad \frac{}{\texttt{(}\textbf{lambda}\texttt{ x a)} \Downarrow \lambda x.a} \qquad \frac{\langle a, \Gamma\rangle \Downarrow i}{\langle \texttt{(}\textbf{def}\texttt{ x a)}, \Gamma\rangle \Downarrow \langle i, \Gamma(x) = i\rangle}\quad \text{(Def)}$$

$$\text{(Macro)}\quad \frac{\langle a, \Gamma\rangle \Downarrow i}{\langle \texttt{(}\textbf{macro}\texttt{ a e)}, \Gamma\rangle \Downarrow i(e)} \qquad \frac{}{\langle a, \Gamma\rangle \Downarrow \Gamma(a)}\quad \text{(Symbol)}$$

Figure 2: The natural semantics of M.

2.1 Def

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.

2.2 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 $\bot$.

2.3 Lambda

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

2.4 Do

Do expressions are expressions of the form `(`**`do`**` value)` They evaluate to a process π`value`. When applied, they apply their `argument` to `value`. If the result of application is not a process, they evaluate to $\bot$ instead.

2.5 Apply

Apply expressions are expressions of the form `(value argument)`. If `value` is a function, it performs application as described in section 2.3. If `value` is a process, it performs application as described in section 2.4.

2.6 Macro

Macro expressions are expressions of the form `(`**`macro`**` name expression)`. Before evaluation, they evaluate `name` and apply it to `expression`, replacing themselves with the result. The `expression` is encoded as a coproduct of a symbol and a list as described in section 3.

3 Encodings

```
;;; Functions

(def id (lambda x x))

(def compose
  (lambda f g
    (lambda x
      (f g x))))

(def const
  (lambda x
    (lambda "" x)))

;;; Booleans

(def true (id const))
(def false (const id))

;;; Products

(def pair
  (lambda first second
    (lambda value
      (value first second))))

(def first
  (lambda pair
    (pair true)))

(def second
  (lambda pair
    (pair false)))

;;; Coproducts

(def left
  (lambda value
    (lambda first ""
      (first value))))

(def right
  (lambda value
    (lambda "" second
      (second value))))

(def left?
  (lambda x
    (x (const true) (const false))))

(def right?
  (lambda x
    (x (const false) (const true))))
```

```
;;; Lists

(def nil (left false))
(def cons (compose right pair))

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

(def cdr
  (lambda list
    (list id right)))

(def nil? left?)

;;; Natural Numbers

(def 0 ())
(def 0? nil?)
(def inc (cons ()))
(def dec cdr)

;;; 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

()