

# Macrolop Specification

Temirkhan Myrzamadi  
e-mail: [hirrolot@gmail.com](mailto:hirrolot@gmail.com)

December 10, 2020

## Abstract

This paper formally describes the form and execution of metaprograms written in Macrolop, an embedded metalanguage aimed at language-oriented programming in C. See also the official repository [1] for the user-friendly overview and the accompanied standard library [2].

## Contents

<b>1</b>	<b>EBNF Grammar</b>	<b>2</b>
<b>2</b>	<b>Reduction Semantics</b>	<b>2</b>
<b>3</b>	<b>Caveats</b>	<b>5</b>

# 1 EBNF Grammar

```

<eval> ::= "MACROLOP_EVAL(" { <term> }+ ")" ;

<term> ::= "call(" <op> "," { <term> }* ")"
        | "v(" <preprocessor-token-list> ")" ;

<op>    ::= <ident> | { <term> }+ ;

```

**Figure 1:** Grammar rules

A metaprogram in Macrolop consists of a non-empty sequence of terms, each of which is either a macro call or just a value.

Notes:

- The grammar above describes metaprograms already expanded by the C pre-processor, except for `MACROLOP_EVAL`, `call`, and `v`.
- `call` accepts `op` either as an identifier or as a non-empty sequence of terms that reduces to an identifier.
- `call` accepts arguments without a separator. This is intentional: it lets arguments be generated programmatically without worrying about separators (for instance, ensuring that there's no separator after the last argument).

However, the `call` syntax hurts IDE support: bad code formatting, no parameters documentation highlighting, et cetera. The workaround is to define a wrapper around an implementation macro like this:

```

#define F00(a, b, c) F00_REAL(a b c)
#define F00_REAL(a, b, c) // The actual implementation here.

```

Then `F00` can be called as `F00(v(1), v(2), v(3))`.

All the public `std`'s macros follow this convention, and moreover, `std`'s public higher-order macros require so for supplied user macros.

## 2 Reduction Semantics

We define reduction semantics for Macrolop. The abstract machine executes configurations of the form  $\langle K; A; C \rangle$ :

- $K$  is a continuation of the form  $\langle K; A; C \rangle$ , where  $C$  includes the `?` sign denoting a result passed into a continuation. For example: let  $K = \langle K'; (1, 2, 3); v(abc) ? \rangle$ , then  $K(v(ghi))$  is  $\langle K'; (1, 2, 3); v(abc) v(ghi) \rangle$ . A special continuation *halt* terminates the abstract machine with provided result.
- $A$  is an accumulator, a sequence 1 of already computed results.
- $C$  (control) is a concrete sequence 3 of terms upon which the abstract machine is operating right now. For example: `call(F00, v(123) v(456)) v(w 8) v(blah)`.

And here are the computational rules:

$(v) : \langle K; A; v(\overline{tok}) \ t \ \overline{t'} \rangle$	$\rightarrow \langle K; A, \ \overline{tok}; t \ \overline{t'} \rangle$
$(v\text{-end}) : \langle K; A; v(\overline{tok}) \rangle$	$\rightarrow K(\text{unseq}(A, \overline{tok}))$
$(op) : \langle K; A; call(\overline{t}, \overline{a}) \ \overline{t'} \rangle$	$\rightarrow \langle \langle K; A; call(?, \overline{a}) \ \overline{t'} \rangle; () ; \overline{t} \rangle$
$(args) : \langle K; A; call(ident, \overline{a}) \ \overline{t} \rangle$	$\rightarrow \langle \langle K; A; ident(?) \ \overline{t} \rangle; () ; \text{comma-sep}(\overline{a}) \rangle$
$(start) : \text{MACROLOP\_EVAL}(t \ \overline{t'})$	$\rightarrow \langle \text{halt}; () ; t \ \overline{t'} \rangle$

**Figure 2:** Computational rules

**Notation 1 (Sequences)**

1. A sequence has the form  $(x_1, \dots, x_n)$ .
2.  $()$  denotes the empty sequence.
3. An element can be appended by comma: if  $a = (1, 2, 3)$  and  $b = 4$ , then  $a, b = (1, 2, 3, 4)$ .
4. **unseq** extracts elements from a sequence without a separator:  
 $\text{unseq}((a, \ b, \ c)) = a \ b \ c$ .

**Notation 2 (Reduction step)**

$\rightarrow$  denotes a single step of reduction (computation).

**Notation 3 (Concrete sequence)**

$\overline{x}$  denotes a concrete sequence  $x_1 \dots x_n$ . For example:  $v(\overline{abc}) \ call(\overline{FOO}, \ v(\overline{123})) \ v(\overline{u \ 8 \ 9})$ .

**Notation 4 (Meta-variables)**

$\overline{tok}$	<i>C preprocessor token</i>
$\overline{ident}$	<i>C preprocessor identifier</i>
$\overline{t}$	<i>Macrolop term</i>
$\overline{a}$	<i>Macrolop term used as an argument</i>

Notes:

- Look at  $(args)$ . Macrolop generates a usual C-style macro invocation with fully evaluated arguments, which will be then expanded by the C preprocessor, resulting in yet another concrete sequence of Macrolop terms to be evaluated by the computational rules.

Therefore, an expansion of  $call(\overline{t}, \overline{a})$  must match the Macrolop grammar, otherwise it might result in a compilation error.

- With the current implementation, at most  $2^{14}$  reduction steps are possible. After exceeding this limit, compilation will likely fail.

The rules are fairly simple: a concrete sequence of terms provided into `MACROLOP_EVAL` is evaluated sequentially till the end; a function's arguments are evaluated before the function is applied, e.g. Macrolop follows applicative evaluation strategy. When there's no more terms to evaluate, the result is pasted where `MACROLOP_EVAL` has been invoked.

The essence of the Macrolop metalanguage is that it allows recursive macro calls. But to be precise, it allows only indirect recursion. Consider these two cases:

- Direct recursion:  $call(X, \overline{tok}) \rightarrow \bar{t}$ , where  $\bar{t}$  contains  $X$ . Then this  $X$  will be blocked forever due to the rules of the C preprocessor (an expansion of  $X(\dots)$  containing  $X$ ).
- Indirect recursion:  $call(X, \bar{a}) \rightarrow \bar{t} \text{ call}(Y, \bar{a}') \bar{t}'$  and  $call(Y, \bar{a}') \rightarrow \bar{t}''$ , where  $\bar{t}''$  contains  $X$ . Then this  $X$  will **not** be blocked by the C preprocessor, e.g. can be invoked again.

**Notation 5 (Multiple reduction steps)**

$\rightarrow$  denotes one or more single evaluation steps, e.g.  $\bar{t} \rightarrow \bar{t}'$  is the same as  $\bar{t} \rightarrow \dots \rightarrow \bar{t}'$ .

Now let's move to the examples of reduction. Take the following code:

```
#define X(op)          call(op, v(123))
#define CALL_X(_123) call(X, v(ID))
#define ID(x)          v(x)
```

See how `call(X, v(CALL_X))` is evaluated:

### Example 1 (Evaluation of terms)

$$\begin{aligned}
& \text{MACROLOP\_EVAL}(\text{call}(X, v(\text{CALL\_X}))) \\
& \quad \downarrow (\text{start}) \\
& \langle \text{halt}; () ; \text{call}(X, v(\text{CALL\_X})) \rangle \\
& \quad \downarrow (\text{args}) \\
& \langle \langle \text{halt}; () ; X(?) \rangle ; () ; v(\text{CALL\_X}) \rangle \\
& \quad \downarrow (v\text{-end}) \\
& \langle \text{halt}; () ; \text{call}(\text{CALL\_X}, v(123)) \rangle \\
& \quad \downarrow (\text{args}) \\
& \langle \langle \text{halt}; () ; \text{CALL\_X}(?) \rangle ; () ; v(123) \rangle \\
& \quad \downarrow (v\text{-end}) \\
& \langle \text{halt}; () ; \text{call}(X, v(ID)) \rangle \langle \\
& \quad \downarrow (\text{args}) \\
& \langle \langle \text{halt}; () ; X(?) \rangle ; () ; v(ID) \rangle \\
& \quad \downarrow (v\text{-end}) \\
& \langle \text{halt}; () ; \text{call}(ID, v(123)) \rangle \\
& \quad \downarrow (\text{args}) \\
& \langle \langle \text{halt}; () ; ID(?) \rangle ; () ; v(123) \rangle \\
& \quad \downarrow (v\text{-end}) \\
& \langle \text{halt}; () ; v(123) \rangle \\
& \quad \downarrow (v\text{-end}) \\
& \text{halt}(123)
\end{aligned}$$

The analogous version written in ordinary C looks like this:

```

#define X(op)      op(123)
#define CALL_X(_123) X(ID)
#define ID(x)      x

```

However, unlike the Macrolop version above, it gets blocked due to the second call to X:

$$X(\text{CALL\_X}) \rightarrow \text{CALL\_X}(123) \rightarrow X(ID)$$

## 3 Caveats

- Consider this scenario:
  - You call F00(1, 2, 3)
  - It gets expanded by the C preprocessor (not by Macrolop)
  - Its expansion contains F00

Then F00 gets blocked by the C preprocessor, e.g. Macrolop cannot handle ordinary macro recursion; you must use `call` to be sure that recursive calls will behave as expected.

I therefore recommend to use only primitive C-style macros (e.g. for performance reasons or because of you cannot express them in Macrolop).

## References

- [1] Temirkhan Myrzamadi. *Language-oriented programming in C*. URL: <https://github.com/Hirrolot/macrolop>.
- [2] Temirkhan Myrzamadi. *The Macrolop standard library documentation*. URL: <https://hirrolot.github.io/macrolop/>.