

$\lambda^{\alpha}\mathcal{M}^{\alpha}$ bytecode reference

Mnemonic	Encoding
BINOP +	01
<i>Adds two integers, with wraparound.</i> −2 , +1	
BINOP −	02
<i>Subtracts two integers, with wraparound.</i> −2 , +1	
BINOP *	03
<i>Multiplies two integers, with wraparound.</i> −2 , +1	
BINOP /	04
<i>Divides two integers, with wraparound. The result is rounded towards zero. The quotient is negative if exactly one operand is negative.</i> −2 , +1	
<i>Raises an error if the divisor is 0.</i>	
BINOP %	05
<i>Computes an integer remainder, with wraparound. The operation satisfies $(a / b) * b + (a \% b) = a$. The remainder is negative if the first operand is negative.</i> −2 , +1	
<i>Raises an error if the divisor is 0.</i>	
BINOP <	06
<i>Tests if the left operand is less than the right operand.</i> −2 , +1	
BINOP <=	07
<i>Tests if the left operand is less than or equal to the right operand.</i> −2 , +1	
BINOP >	08
<i>Tests if the left operand is greater than the right operand.</i> −2 , +1	
BINOP >=	09
<i>Tests if the left operand is greater than or equal to the right operand.</i> −2 , +1	
BINOP ==	0a
<i>Tests if the left operand is equal to the right operand. One of the operands must be an integer. Integers are never equal to values of other types.</i> −2 , +1	
BINOP !=	0b
<i>Tests if the left operand is not equal to the right operand. Unlike ==, both operands must be integers.</i> −2 , +1	
BINOP &&	0c
<i>Tests if both integer operands are non-zero.</i> −2 , +1	
BINOP !!	0d
<i>Tests if either of the operands is non-zero.</i> −2 , +1	
CONST <i>k</i>	10 [<i>k</i> : i32]
<i>Pushes a constant immediate <i>k</i> onto the stack as an integer value.</i> −0 , +1	
STRING <i>s</i>	11 [<i>s</i> : i32]
<i>Pushes a string starting at offset <i>s</i> in the string table onto the stack.</i> −0 , +1	
SEXP <i>s n</i>	12 [<i>s</i> : i32] [<i>n</i> : i32]
<i>Constructs an S-expression with <i>n</i> members. A string starting at offset <i>s</i> in the string table is used as the tag.</i> −<i>n</i> , +1	
STI	13
<i>Performs an indirect store to a variable. The first operand must be a reference to the variable. The second operand is assigned to the variable.</i> −2 , +1	
<i>Pushes the second operand back onto the stack (for chained assignments).</i>	
STA	14
<i>Performs an indirect store to a variable or an aggregate. The operation is overloaded; its behavior depends on the second-to-top value on the stack, which must be either a reference to a variable or an integer:</i>	
<ul style="list-style-type: none"> If its type is a reference to a variable, this operation is equivalent to STI. In particular, it pops 2 operands and pushes 1. If its type is an integer, this operations pops 3 operands and pushes 1. The first operand must be an aggregate: an S-expression, an array, or a string. The second operand (the integer) is 	

Mnemonic	Encoding
<i>an index into the aggregate. The third operand is assigned to the aggregate at this index.</i>	
<i>The index must fall within the range from 0 (inclusive) to <i>l</i> (exclusive), where <i>l</i> is the length of the aggregate. Raises an error if the index is outside the bounds.</i>	
<i>Pushes the third operand back onto the stack (for chained assignments).</i>	
JMP <i>l</i>	15 [<i>l</i> : i32]
<i>Sets the instruction counter to <i>l</i>.</i> −0 , +0	
END	16
<i>Marks the end of the procedure definition. When executed, returns the top value to the caller of this procedure.</i> −1 , +1	
RET	17
<i>Returns the top value to the caller of this procedure.</i> −1 , +1	
DROP	18
<i>Removes the top value from the stack.</i> −1 , +0	
DUP	19
<i>Duplicates the top value of the stack.</i> −1 , +2	
SWAP	1a
<i>Swaps the top two values on the stack.</i> −2 , +2	
ELEM	1b
<i>Looks up an element of an aggregate by its index. The first operand must be the aggregate: an S-expression, an array, or a string. The second operand must be an integer, taken as an index into the aggregate.</i> −2 , +1	
<i>The index must fall within the range from 0 (inclusive) to <i>l</i> (exclusive), where <i>l</i> is the length of the aggregate. Raises an error if the index is outside the bounds.</i>	
LD <i>G</i> (<i>m</i>)	20 [<i>m</i> : i32]
<i>Pushes the <i>m</i>th global onto the stack.</i> −0 , +1	
LD <i>L</i> (<i>m</i>)	21 [<i>m</i> : i32]
<i>Pushes the <i>m</i>th local onto the stack.</i> −0 , +1	
LD <i>A</i> (<i>m</i>)	22 [<i>m</i> : i32]
<i>Pushes the <i>m</i>th function argument onto the stack.</i> −0 , +1	
LD <i>C</i> (<i>m</i>)	23 [<i>m</i> : i32]
<i>Pushes the <i>m</i>th variable captured by this closure onto the stack.</i> −0 , +1	
LDA <i>G</i> (<i>m</i>)	30 [<i>m</i> : i32]
<i>Pushes a reference to the <i>m</i>th global onto the stack.</i> −0 , +1	
LDA <i>L</i> (<i>m</i>)	31 [<i>m</i> : i32]
<i>Pushes a reference to the <i>m</i>th local onto the stack.</i> −0 , +1	
LDA <i>A</i> (<i>m</i>)	32 [<i>m</i> : i32]
<i>Pushes a reference to the <i>m</i>th function argument onto the stack.</i> −0 , +1	
LDA <i>C</i> (<i>m</i>)	33 [<i>m</i> : i32]
<i>Pushes a reference to the <i>m</i>th variable captured by this closure onto the stack.</i> −0 , +1	
ST <i>G</i> (<i>m</i>)	40 [<i>m</i> : i32]
<i>Stores a value in the <i>m</i>th global. Pushes the value back onto the stack (for chained assignments).</i> −1 , +1	
ST <i>L</i> (<i>m</i>)	41 [<i>m</i> : i32]
<i>Stores a value in the <i>m</i>th local. Pushes the value back onto the stack (for chained assignments).</i> −1 , +1	
ST <i>A</i> (<i>m</i>)	42 [<i>m</i> : i32]
<i>Stores a value in the <i>m</i>th function argument. Pushes the value back onto the stack (for chained assignments).</i> −1 , +1	
ST <i>C</i> (<i>m</i>)	43 [<i>m</i> : i32]
<i>Stores a value in the <i>m</i>th variable captured by this closure. Pushes the value back onto the stack (for chained assignments).</i> −1 , +1	
CJMPz <i>l</i>	50 [<i>l</i> : i32]
<i>Sets the instruction pointer to <i>l</i> if the operand is zero. Otherwise, moves to the next instruction.</i> −1 , +0	

Mnemonic	Encoding
CJMPnz <i>l</i>	51 [<i>l</i> : i32]
<i>Sets the instruction pointer to <i>l</i> if the operand is non-zero. Otherwise, moves to the next instruction.</i> −1 , +0	
BEGIN <i>a n</i>	52 [<i>a</i> : i32] [<i>n</i> : i32]
<i>Marks the start of a procedure definition with a arguments and <i>n</i> locals. When executed, initializes locals to an empty value.</i> −0 , +0	
<i>Unlike CBEGIN, the defined procedure cannot use captured variables.</i>	
CBEGIN <i>a n</i>	53 [<i>a</i> : i32] [<i>n</i> : i32]
<i>Marks the start of a closure definition with a arguments and <i>n</i> locals. When executed, initializes locals to an empty value.</i> −0 , +0	
<i>Unlike BEGIN, the defined closure may use captured variables.</i>	
CLOSURE <i>l n</i> <i>V</i> ₁ (<i>m</i> ₁) ... <i>V</i> _{<i>n</i>} (<i>m</i> _{<i>n</i>})	54 [<i>l</i> : i32] [<i>n</i> : i32] [[<i>V</i> _{<i>i</i>} (<i>m</i> _{<i>i</i>}): varspec; <i>n</i>] varspec immediates are encoded as follows: • <i>G</i> (<i>m</i>): 00 [<i>m</i> : i32] • <i>L</i> (<i>m</i>): 01 [<i>m</i> : i32] • <i>A</i> (<i>m</i>): 02 [<i>m</i> : i32] • <i>C</i> (<i>m</i>): 03 [<i>m</i> : i32]
<i>Pushes a new closure with <i>n</i> captured variables onto the stack. The bytecode for the closure begins at <i>l</i> (given as an offset from the start of the bytecode).</i> −0 , +1	
<i>The instruction has a variable-length encoding; the description of each captured variable is specified as a 5-byte immediate.</i>	
CALLC <i>n</i>	55
<i>Calls a closure with <i>n</i> arguments. The first operand must be the closure, followed by the arguments. Pushes the returned value onto the stack.</i> −(<i>n</i> + 1) , +1	
CALL <i>l n</i>	56 [<i>l</i> : i32] [<i>n</i> : i32]
<i>Calls a function with <i>n</i> arguments. The bytecode for the function begins at <i>l</i> (given as an offset from the start of the bytecode). Pushes the returned value onto the stack.</i> −<i>n</i> , +1	
<i><i>l</i> must not refer to a closure definition (declared with CBEGIN) because this instruction does not capture any variables.</i>	
TAG <i>s n</i>	57 [<i>s</i> : i32] [<i>n</i> : i32]
<i>Tests whether the operand is an S-expression with a specific tag (the string starting at offset <i>s</i> in the string table) and number of elements (<i>n</i>).</i> −1 , +1	
<i>If the operand is not an S-expression, pushes 0.</i>	
ARRAY <i>n</i>	58 [<i>n</i> : i32]
<i>Tests whether the operand is an array of <i>n</i> elements.</i> −1 , +1	
FAIL ln col	59 [ln: i32] [col: i32]
<i>Raises an error, reporting a match failure at line ln, column col (both 1-based). The operand is the value being matched.</i> −1 , +1	
LINE ln	5a [ln: i32]
<i>Marks the following bytecode as corresponding to line ln in the source text. Only used for diagnostics.</i> −0 , +0	
PATT =str	60
<i>Tests whether the two operands are both strings and store the same bytes.</i> −2 , +1	
PATT #string	61
<i>Tests whether the operand is a string.</i> −1 , +1	
PATT #array	62
<i>Tests whether the operand is an array.</i> −1 , +1	
PATT #sexp	63
<i>Tests whether the operand is an S-expression.</i> −1 , +1	
PATT #ref	64
<i>Tests whether the operand has a boxed representation (passed by reference).</i> −1 , +1	
PATT #val	65
<i>Tests whether the operand has an unboxed representation (passed by value).</i> −1 , +1	

Mnemonic	Encoding
PATT #fun	66
<i>Tests whether the operand is a closure.</i> −1 , +1	
CALL Lread	70
<i>Calls the built-in function read. The function returns the next program input. If the program input is exhausted, raises an error.</i> −0 , +1	
<i>Consecutive calls to read returns consecutive inputs.</i>	
CALL Lwrite	71
<i>Calls the built-in function write. The operand must be an integer. The function adds the operand to the program output. Returns an empty value.</i> −1 , +1	
CALL Llength	72
<i>Calls the built-in function length. The operand must be an aggregate: an S-expression, an array, or a string. The function returns the length of the aggregate as an integer.</i> −1 , +1	
CALL Lstring	73
<i>Calls the built-in function string. The operand must be an integer, a string, an array, or an S-expression. If the operand is an array or an S-expression, the type requirements apply transitively to the operand's elements. The function returns a string representation of the operand.</i> −1 , +1	
CALL Barray <i>n</i>	74 [<i>n</i> : i32]
<i>Calls the built-in function .array. The function creates an array composed of the <i>n</i> operands and returns it.</i> −<i>n</i> , +1	

Notation

- Literary bytes in the encoding are written in hexadecimal. Integer immediates are encoded as signed numbers in native endianness.
- The number in red tells how many operands the operation pops off the stack. The number in green indicates how many values it then pushes onto the stack.

Notes

- Arithmetic is performed modulo 2³¹ on 32-bit platforms and 2⁶³ on 64-bit platforms. All operations are signed.
- Boolean values (resulting from comparisons) are represented as integers: 1 if true, 0 if false. For logical operations, a non-zero integer value is true.
- Operands are ordered from the lowest up; the rightmost operand is on the top.
- Operations perform type-checking dynamically, raising an error if an operand has an unexpected type.
- Jump targets are byte offsets from the start of the bytecode. In other words, all jumps are absolute.
- A closure is a procedure that captures variables in outer scopes. Variables are captured by-value, not by-reference. A closure with no captured values can be called as a regular procedure (see CALL).
- This reference assumes that no values introduced by a procedure (other than a value to be returned) remain on the stack when RET or END is executed – that is, the stack height at exit points is larger than at the entry by exactly one element.
- Integers are always stored unboxed and passed by value. All other types are always boxed and passed by reference.