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

Revision 6

Mnemonic	Encoding	
`BINOP +`	`01`	
Adds two integers, with wraparound.		-2, $+1$
`BINOP -`	`02`	
Subtracts two integers, with wraparound.		-2, $+1$
`BINOP *`	`03`	
Multiplies two integers, with wraparound.		-2, $+1$
`BINOP /`	`04`	
Divides two integers, with wraparound. The result is rounded towards zero. The quotient is negative if exactly one operand is negative. *Raises an error if the divisor is 0.*		-2, $+1$
`BINOP %`	`05`	
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.* *Raises an error if the divisor is 0.*		-2, $+1$
`BINOP <`	`06`	
Tests if the left operand is less than the right operand.		-2, $+1$
`BINOP <=`	`07`	
Tests if the left operand is less than or equal to the right operand.		-2, $+1$
`BINOP >`	`08`	
Tests if the left operand is greater than the right operand.		-2, $+1$
`BINOP >=`	`09`	
Tests if the left operand is greater than or equal to the right operand.		-2, $+1$
`BINOP ==`	`0a`	
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.		-2, $+1$
`BINOP !=`	`0b`	
Tests if the left operand is not equal to the right operand. Unlike `==`*, both operands must be integers.*		-2, $+1$
`BINOP &&`	`0c`	
Tests if both integer operands are non-zero.		-2, $+1$
`BINOP !!`	`0d`	
Tests if either of the operands is non-zero.		-2, $+1$
`CONST` k	`10 [`k`: i32]`	
Pushes a constant immediate k onto the stack as an integer value.		-0, $+1$
`STRING` s	`11 [`s`: i32]`	
Pushes a string starting at offset s in the string table onto the stack.		-0, $+1$
`SEXP` s n	`12 [`s`: i32] [`n`: i32]`	
Constructs an S-expression with n members. A string starting at offset s in the string table is used as the tag.		$-n$, $+1$
`STI`	`13`	
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. *Pushes the second operand back onto the stack (for chained assignments).* ***Note:** this instruction is never emitted by the Lama compiler.*		-2, $+1$
`STA`	`14`	
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:* • *If its type is a reference to a variable, this operation is equivalent to* `STI`*. In particular, it pops 2 operands and pushes 1.* ***Note:** the bytecode emitted by the Lama compiler never triggers this case as it does not contain* `LDA` *instructions.* • *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 an index into the aggregate. The third operand is assigned to the aggregate at this index.* *The index must fall within the range from 0 (inclusive) to l (exclusive), where l is the length of the aggregate. Raises an error if the index is outside the bounds.* *Pushes the third operand back onto the stack (for chained assignments).*		
`JMP` l	`15 [`l`: i32]`	
Sets the instruction counter to l.		-0, $+0$
`END`	`16`	
Marks the end of the procedure definition. When executed, returns the top value to the caller of this procedure.		-1, $+1$
`RET`	`17`	
Returns the top value to the caller of this procedure.		-1, $+1$
`DROP`	`18`	
Removes the top value from the stack.		-1, $+0$
`DUP`	`19`	
Duplicates the top value of the stack.		-1, $+2$
`SWAP`	`1a`	
Swaps the top two values on the stack.		-2, $+2$
`ELEM`	`1b`	
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. *The index must fall within the range from 0 (inclusive) to l (exclusive), where l is the length of the aggregate. Raises an error if the index is outside the bounds.*		-2, $+1$
`LD G(`m`)`	`20 [`m`: i32]`	
Pushes the mth global onto the stack.		-0, $+1$
`LD L(`m`)`	`21 [`m`: i32]`	
Pushes the mth local onto the stack.		-0, $+1$
`LD A(`m`)`	`22 [`m`: i32]`	
Pushes the mth function argument onto the stack.		-0, $+1$
`LD C(`m`)`	`23 [`m`: i32]`	
Pushes the mth variable captured by this closure onto the stack.		-0, $+1$
`LDA G(`m`)`	`30 [`m`: i32]`	
Pushes a reference to the mth global onto the stack. ***Note:** this instruction is never emitted by the Lama compiler.*		-0, $+1$
`LDA L(`m`)`	`31 [`m`: i32]`	
Pushes a reference to the mth local onto the stack. ***Note:** this instruction is never emitted by the Lama compiler.*		-0, $+1$
`LDA A(`m`)`	`32 [`m`: i32]`	
Pushes a reference to the mth function argument onto the stack. ***Note:** this instruction is never emitted by the Lama compiler.*		-0, $+1$
`LDA C(`m`)`	`33 [`m`: i32]`	
Pushes a reference to the mth variable captured by this closure onto the stack. ***Note:** this instruction is never emitted by the Lama compiler.*		-0, $+1$
`ST G(`m`)`	`40 [`m`: i32]`	
Stores a value in the mth global. Pushes the value back onto the stack (for chained assignments).		-1, $+1$
`ST L(`m`)`	`41 [`m`: i32]`	
Stores a value in the mth local. Pushes the value back onto the stack (for chained assignments).		-1, $+1$
`ST A(`m`)`	`42 [`m`: i32]`	
Stores a value in the mth function argument. Pushes the value back onto the stack (for chained assignments).		-1, $+1$
`ST C(`m`)`	`43 [`m`: i32]`	
Stores a value in the mth variable captured by this closure. Pushes the value back onto the stack (for chained assignments).		-1, $+1$
`CJMPz` l	`50 [`l`: i32]`	
Sets the instruction pointer to l if the operand is zero. Otherwise, moves to the next instruction.		-1, $+0$
`CJMPnz` l	`51 [`l`: i32]`	
Sets the instruction pointer to l if the operand is non-zero. Otherwise, moves to the next instruction.		-1, $+0$
`BEGIN` a n	`52 [`a`: i32] [`n`: i32]`	
Marks the start of a procedure definition with a arguments and n locals. When executed, initializes locals to an empty value. *Unlike* `CBEGIN`*, the procedure cannot use captured variables.*		-0, $+0$
`CBEGIN` a n	`53 [`a`: i32] [`n`: i32]`	
Marks the start of a closure definition with a arguments and n locals. When executed, initializes locals to an empty value. *Unlike* `BEGIN`*, the defined closure may use captured variables.*		-0, $+0$

`CLOSURE` l n $V_1(m_1)$ … $V_n(m_n)$	`54 [`l`: i32] [`n`: i32]`
	`[[`$V_i(m_i)$`: varspec]; `n`]`
	`varspec` immediates are encoded as follows:
	• `G(`m`): 00 [`m`: i32]`
	• `L(`m`): 01 [`m`: i32]`
	• `A(`m`): 02 [`m`: i32]`
	• `C(`m`): 03 [`m`: i32]`	

Pushes a new closure with n captured variables onto the stack. The bytecode for the closure begins at l (given as an offset from the start of the bytecode). *The instruction has a variable-length encoding; the description of each captured variable is specified as a 5-byte immediate.*		-0, $+1$
`CALLC` n	`55`	
Calls a closure with n arguments. The first operand must be the closure, followed by the arguments. Pushes the returned value onto the stack.		$-(n+1)$, $+1$
`CALL` l n	`56 [`l`: i32] [`n`: i32]`	
Calls a function with n arguments. The bytecode for the function begins at l (given as an offset from the start of the bytecode). Pushes the returned value onto the stack. *l must not refer to a closure definition (declared with* `CBEGIN`*) because this instruction does not capture any variables.*		$-n$, $+1$
`TAG` s n	`57 [`s`: i32] [`n`: i32]`	
Tests whether the operand is an S-expression with a specific tag (the string starting at offset s in the string table) and number of elements (n). *If the operand is not an S-expression, pushes 0.*		-1, $+1$
`ARRAY` n	`58 [`n`: i32]`	
Tests whether the operand is an array of n elements.		-1, $+1$
`FAIL` ln col	`59 [ln: i32] [col: i32]`	
Raises an error, reporting a match failure at line ln, column col (both 1-based). The operand is the value being matched.		-1, $+!$
`LINE` ln	`5a [ln: i32]`	
Marks the following bytecode as corresponding to line ln in the source text. Only used for diagnostics.		-0, $+0$
`PATT =str`	`60`	
Tests whether the two operands are both strings and store the same bytes.		-2, $+1$
`PATT #string`	`61`	
Tests whether the operand is a string.		-1, $+1$
`PATT #array`	`62`	
Tests whether the operand is an array.		-1, $+1$
`PATT #sexp`	`63`	
Tests whether the operand is an S-expression.		-1, $+1$
`PATT #ref`	`64`	
Tests whether the operand has a boxed representation (passed by reference).		-1, $+1$
`PATT #val`	`65`	
Tests whether the operand has an unboxed representation (passed by value).		-1, $+1$
`PATT #fun`	`66`	
Tests whether the operand is a closure.		-1, $+1$
`CALL Lread`	`70`	
Calls the built-in function `read`*. The function returns the next program input. If the program input is exhausted, raises an error.* *Consecutive calls to* `read` *returns consecutive inputs.*		-0, $+1$
`CALL Lwrite`	`71`	
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.*		-1, $+1$
`CALL Llength`	`72`	
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.*		-1, $+1$
`CALL Lstring`	`73`	
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.*		-1, $+1$
`CALL Barray` n	`74 [`n`: i32]`	
Calls the built-in function `.array`*. The function creates an array composed of the n operands and returns it.*		$-n$, $+1$

Notation

1. Literal bytes in the encoding are written in hexadecimal. Integer immediates are encoded as signed numbers in native endianness.
2. 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

1. Arithmetic is performed modulo 2^{31} on 32-bit platforms and 2^{63} on 64-bit platforms. All operations are signed.
2. 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.
3. Operands are ordered from the lowest up; the rightmost operand is on the top.
4. Operations perform type-checking dynamically, raising an error if an operand has an unexpected type.
5. Jump targets are byte offsets from the start of the bytecode. In other words, all jumps are absolute.
6. 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`).
7. 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.
8. Integers are always stored unboxed and passed by value. All other types are always boxed and passed by reference.