Specification of `enkel0`

Virtual Machine Specification

The `enkel0` virtual machine (VM) is a stack-based machine designed to execute programs written in the enkel0 programming language. It supports basic arithmetic operations, control flow instructions, function calls, and I/O operations.

Features

- Stack-based architecture with a fixed-size stack.
- Arithmetic operations: `ADD`, `SUB`, `MUL`, `DIV`, `MOD`.
- Logical operations: `AND`, `OR`, `XOR`.
- Comparison operations: `EQ`, `NEQ`, `GT`, `LT`, `GQ`, `LQ`.
- Control flow: `JMP`, `JPZ`, `JPNZ` for unconditional and conditional jumps.
- Function calls and returns with `CALL` and `RET` instructions.
- Memory access: `LD`, `ST`, `LOAD`, `STORE` for variables and arrays.
- Input/output operations: `PRINT` and `EMIT` for printing to console.
- Error handling: Runtime error for division by zero.

Instruction Set

Arithmetic Instructions

- `ADD` : Add top two elements of stack
- `SUB` : Subtract top element from next-to-top element
- `MUL` : Multiply top two elements of stack
- `DIV` : Divide next-to-top element by top element
- `MOD` : Compute modulus of top two elements

Logical Instructions

- `AND` : Bitwise AND of top two elements

- `OR` : Bitwise OR of top two elements
- `XOR` : Bitwise XOR of top two elements
- `EQ` : Compare top two elements for equality
- `NEQ` : Compare top two elements for inequality
- `GT` : Compare top two elements (greater than)
- `LT` : Compare top two elements (less than)
- `GQ` : Compare top two elements (greater than or equal)
- `LQ` : Compare top two elements (less than or equal)

Control Instructions

- `JMP addr` : Unconditionally jump to instruction at addr
- `JPZ addr` : Jump to addr if top of stack is zero
- `JPNZ addr` : Jump to addr if top of stack is not zero
- `CALL addr` : Call function at addr
- `RET` : Return from function call

Memory Instructions

- `LD offset` : Load value from local variable at offset from frame pointer
- `ST offset` : Store value into local variable at offset from frame pointer
- `LOAD addr` : Load value from global variable at addr
- `STORE addr` : Store value into global variable at addr

I/O Instructions

- `PRINT` : Print top value of stack as integer followed by newline
- `EMIT` : Print top value of stack as character

Miscellaneous Instructions

- `NOP` : No operation
- `HALT` : Halt execution

Memory Layout

- **Stack**: Used for operand storage and temporary storage during execution.
- **Variables**: Arrays for storing local and global variables.
- **Arguments**: Array for function arguments.

- **Arrays**: Array for storing dynamically allocated arrays (not fully implemented in this specification).

Implementation Details[1]

The `enkel/0 VM` is implemented in C with the following components:

- `VM`: Struct to hold VM state including stack, code, program counter, frame pointer, and various arrays.
- `newVM`: Function to initialize a new VM instance with provided code, program counter, variable count, argument count, and array count.
- `freeVM`: Function to free memory allocated for a VM instance.
- `push`, `pop`: Functions to push and pop values onto/from the stack.
- `nextcode`: Function to fetch the next instruction from the code segment.
- `run`: Function to execute the code segment until a HALT instruction is encountered.

Language Specification[2]

Grammar

A more formal EBNF-like description of the language `enkel/0`:

```
program = block "." .

block =
    ["const" ident "=" number {"," ident "=" number} ";"]
    ["array" ident "=" number {"," ident "=" number} ";"]
    ["var" ident {"," ident} ";"]
    {"procedure" ident "[" ident {"," ident} "]" ";" block ";"}
    statement .

statement =
    ident ["." index] "is" expression
    | "call" ident "[" ident {"," ident} "]"
    | "begin" statement {";" statement } "end"
    | "if" condition "then" statement { "else" statement }
    | "while" condition "do" statement
    | "do" statement "while" condition
    | return [factor]
    | print factor
    | emit factor .
```

[1]The implementation is seldom part of the specification naturally, but sometime a "reference" implementation can be given.

[2]This confuses *implementation and specification*. But as the implementation here guides us as we work bottom up, it has effected he specification. Later a more proper conception will be reached. This only give you an idea of what is going on.

```
condition =
    expression ("="|"#"|"<"|"<="|">"|">=") expression .

expression = ["-"] term {("+"|"-"|"or"|"xor") term } .

term = factor {("*"|"/"|"%"|"and") factor } .

factor =
    ident ["." index]
    | number
    | "(" expression ")" .

index =
    ident
    | number .
```

Program and Block

If we start with how the structure of a program looks like, it consists of a `<block>` and ends with a period.

The `<block>` may have a `const` definition at the start, an `array`, or global variables `var`. Constants are global and may not be changed, only assigned once at the start. Arrays are also global (and have no checks for out-of-bounds addressing). Global variables may be assigned and reassigned throughout the program.

After the initialization, there is an optional list of procedures. The procedure is recognized by a program unique identifier `<ident>`. Then there is an optional list of arguments that come from the call to the procedure. The arguments values) are copied from the call to the procedure. Each argument `<ident>` is separated with a comma. The arguments work as local variables throughout the procedure. The procedure `<block>` is ended with a semicolon.

Then at last, there are statements that are the first to be called and contain the main code of the program. A program might thus only consist of `<statement>` (in a `<block>`).

Statement

A `<statement>` may consist of:

- An identifier `<ident>`, assigned by `is` to the value of an evaluated `<expression>`. The identifier might be in the form of an array which is treated as "two variables glued together by a period".
- A call with an identifier `<ident>` and possible arguments in the format of `<factor>`, separated by a comma.
- A group of statements starting with `begin` and ending with `end`. The statements themselves are separated by a semicolon. This might be one of the things you might

want to change, e.g., with curly brackets, as in C, instead.

- A conditional jump, with `if` and `then`. This statement may include optional `else`. Thus, if a `<condition>` has been met (true), then the `<statement>` that follows will be executed, else if the condition was false, another `<statement>` might be executed instead.
- Two conditional jump structures are `while <condition> do <statement>`, and the reverse `do <statement> while <condition>`. As long as the `<condition>` is met, the `<statement>` will be executed continuously. The latter `do-while` executes at least one `<statement>`, which the former `while-do` does not.
- A `return` statement with an optional return value. All return values are also copied to a special `rval` global variable. Therefore, `rval` can also be treated as an ordinary global variable, although storing a value will be overwritten as soon as a call with a returning value is made.
- There is a `print` of integer values.
- And `emit` can be used to display an ASCII character.

Condition

The `<condition>` occurs mostly together with some jump instruction, `if-else` statements, `while-do`, or `do-while`. Anything other than 0 is taken as true. It also departs from expressions which are compared with relative operations such as `<` less han or `=` equal. A special symbol `#` stands for "not equal" (other notations could be e.g., `"<>"` or `"!="`).

Expression

The `<expression>` can be a `<term>` with an optional unary minus before. Or it (also) could be addition with a plus sign `+`, or subtractions with a minus sign `-`, or a logical `or` or `xor`.

Term

The `<term>` can be a `<factor>`. It can also be a `<factor>` with operators `*`, `/`, `%` or `and` connecting to the other `<factor>`.

Factor

The `<factor>` takes an ordinary `<ident>` or an identifier with an index, i.e., an array, or it is a `<number>`, or it is an `<expression>` with parentheses around it.

Index

The implementation of arrays has been very restricted to only care for global arrays. In that way, they are easy to handle. Arrays are basically global variables pointing to an array in the VM, with an attached `<index>` which points out where the value is, or the slot that will be assigned with a value. Also, to keep the implementation small only a `<number>` (integer) or an identifier `<ident>` is allowed.

ASCII and emit

In order to get printed text, there is an option of putting out letters each at a time. Using the code table for ASCII we can emit a character at a time.

0	nul	1	soh	2	stx	3	etx	4	eot	5	enq	6	ack	7	bel
8	bs	9	ht	10	nl	11	vt	12	np	13	cr	14	so	15	si
16	dle	17	dc1	18	dc2	19	dc3	20	dc4	21	nak	22	syn	23	etb
24	can	25	em	26	sub	27	esc	28	fs	29	gs	30	rs	31	us
32	sp	33	!	34	"	35	#	36	$	37	%	38	&	39	’
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	‘	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	del

To print ”Hello”, emit ’H’, ’e’, ’l’, ’l’, and ’o’ with a new line at the end (carriage return 13 and line feed 10), a statement can look like:

```
begin emit 72; emit 101; emit 108; emit 108; emit 111; emit 13; emit 10 end
```

Conclusion

The `enkel0` VM provides a basic framework for executing programs written in the `enkel0` programming language. It supports essential operations for arithmetic, logic, control flow, and I/O, making it suitable for educational purposes and small-scale applications (not really).