The STklos Virtual Machine

Jeronimo Pellegrini
# Table of Contents

1. The bytecode .......................................................... 2
2. Value register ......................................................... 3
3. Stack ........................................................................ 6
4. Local variables .......................................................... 7
5. Deep variables ........................................................... 9
6. Global variables .......................................................... 12
   6.1. UGLOBAL_{REF,SET} and the checked global table ....... 12
7. Operations ................................................................. 15
   7.1. Arithmetic ............................................................ 15
   7.2. Increment and decrement val .................................. 17
   7.3. Comparisons ....................................................... 17
   7.4. Constructors ....................................................... 18
   7.5. Structure references ............................................. 19
8. Control flow ............................................................... 21
9. Closures, let, and related ............................................ 22
   9.1. let .................................................................. 22
10. Miscelannea ............................................................... 23
   10.1. Creating closures and procedures ......................... 24
   10.2. Procedure calls ................................................ 25
11. Modules ................................................................. 29
12. vm.c .................................................................. 31
   12.1. The global lock .................................................. 31
   12.2. run_vm(vm_thread *vm) ....................................... 32
13. Continuations ............................................................ 33
This is the documentation for the opcodes of the STklos virtual machine. The VM implementation is contained in the files src/vm.h and src/vm.c.

The VM has a stack, which in the source code is accessed using the C functions push(elt) and pop(). Each VM thread also has:

- STk_instr *pc, the program counter
- SCM *fp, the frame pointer
- SCM *sp, the Scheme stack pointer
- SCM *stack, the Scheme stack
- int stack_len, the length of the stack
- SCM val, a register for the current value
- SCM vals[], a register for multiple values
- int valc, the number of multiple values
- SCM r1, r2 two registers
- SCM env, the current environment
- SCM current_module, the current module
- SCM iport, oport, eport, the current input, output and error ports
- SCM scheme_thread, the Scheme thread associated with this thread

Of these, only a few are relevant to understanding the bytecode – these are the value registers and the stack.
STklos bytecode is a sequence of 16-bit integers. You can see the opcodes of a compiled thunk with

```
(disassemble (lambda () ...))
```

and the opcodes of an expression with

```
(disassemble-expr 'expr)
```

With an extra `#t` argument, `disassemble-proc` will show constants:

```
(disassemble-expr "abc"
```

<table>
<thead>
<tr>
<th>000:</th>
<th>CONSTANT</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>002:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Constants:
- 0: "abc"

When we make a closure with the lambda, we’ll always see a `RETURN` at the end of the output:

```
stklos> (disassemble (lambda () '())
```

| 000: | IM-NIL |
| 001: | RETURN |

In the above example, one opcode loads the `NIL` value to the register and another opcode `RETURN`’s. This return is from the lambda.
Chapter 2. Value register

The simpler opcodes are those that carry with them an immediate value. These operations will copy their value to the val register in the VM.

<table>
<thead>
<tr>
<th>IM_FALSE</th>
<th>IM_TRUE</th>
<th>IM_NIL</th>
<th>IM_MINUS1</th>
<th>IM_ZERO</th>
<th>IM_ONE</th>
<th>IM_VOID</th>
</tr>
</thead>
</table>

Examples:

```scheme
(disassemble-expr 1)
```

```
000: IM-ONE
```

```scheme
(disassemble (lambda () #f 1))
```

```
000: IM-FALSE
001: IM-ONE
002: RETURN
```

Opcodes for small integers and constants do the same, but they take a little longer to execute, since they need to perform some small operations.

<table>
<thead>
<tr>
<th>SMALL_INT</th>
<th>CONSTANT</th>
</tr>
</thead>
</table>

```scheme
(disassemble-expr 5)
```

```
000: SMALL-INT 5
```

Small integers are not the same as fixnums! A small integer is an integer number that fits in 16 bits (that is, in one bytecode element). The fixnum range depends on the size of long in the platform being used.

Suppose STklos has been compiled on a 64 bit system and also on a 32 bit system. The ranges for
small ints and fixnums are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>small integer (on both)</td>
<td>([-2^{15}, +2^{15} - 1])</td>
</tr>
<tr>
<td>fixnum (long is 32-bit)</td>
<td>([-2^{29}, +2^{29} - 1])</td>
</tr>
<tr>
<td>fixnum (long is 64-bit)</td>
<td>([-2^{61}, +2^{61} - 1])</td>
</tr>
</tbody>
</table>

The expression above, 5, is compiled into the bytes

00 08 00 05

where 00 08 is the opcode for `small int', and 00 05 is the argument (the small integer, 5).

Small integers are compiled into the bytecode. Fixnums, bignums, strings are stored outside of the bytecode, and the instruction `CONSTANT` takes as argument an index into the constants vector.

The expression 50000 is not a small integer, so it is compiled as a constant:

```lisp
(disassemble-expr 50000 #t)
000:  CONSTANT             0
002:  
Constants:
  0: 50000
```

Zero is the index of 50000 in the constants vector.

The above code is compiled into bytecode as

00 09 00 00

where 00 09 means `CONSTANT' and 00 00 is the index into the constants vector.

Another clarifying example:

```lisp
(disassemble-expr '(values 50000 'abc) #t)
```

```lisp
  000:  PREPARE-CALL
  001:  CONSTANT-PUSH        0
  003:  CONSTANT-PUSH        1
  005:  GREF-INVOKE          2 2
  008:  
Constants:
  0: 50000
  1: "abc"
```
The bytecode is

```
37 85 0 85 1 86 2 2
```

Here,

- `85 0` is **CONSTANT-PUSH 0** (0 = first element of the vector)
- `85 1` is **CONSTANT-PUSH 1** (1 = second element)
- `86 2 2` is **GREF-INVOKE 2 2** (2 = number, arg to `values`, next 2 = third element of vector)
Chapter 3. Stack

The following opcodes are similar to the immediate-value ones, except that, instead of copying their values to the `val` register, they push the value on the stack.

<table>
<thead>
<tr>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE_PUSH</td>
</tr>
<tr>
<td>TRUE_PUSH</td>
</tr>
<tr>
<td>NIL_PUSH</td>
</tr>
<tr>
<td>MINUS1_PUSH</td>
</tr>
<tr>
<td>ZERO_PUSH</td>
</tr>
<tr>
<td>ONE_PUSH</td>
</tr>
<tr>
<td>VOID_PUSH</td>
</tr>
<tr>
<td>INT_PUSH</td>
</tr>
<tr>
<td>CONSTANT_PUSH</td>
</tr>
</tbody>
</table>

The `POP` and `PUSH` move objects between stack and value register.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>POP</td>
<td>move top of stack to val register</td>
</tr>
<tr>
<td>PUSH</td>
<td>store val register on top of stack</td>
</tr>
</tbody>
</table>
Chapter 4. Local variables

The `LOCAL_REF` opcodes will load the values of variables from the current environment (the `'local'` variables) on the `'val` register.

Examples:

```
(disassemble (lambda (a) a))
000:  LOCAL-REF0
001:  RETURN
```

```
(disassemble (lambda (a b) a))
000:  LOCAL-REF1
001:  RETURN
```

There are opcodes for five fixed positions only, so after that another opcode, `LOCAL_REF`, needs an argument:

```
(disassemble (lambda (a b c d e f) a))
000:  LOCAL-REF 5
002:  RETURN
```

The following opcodes are similar to the local reference ones, except that, instead of copying their values to the `val` register, they push the value on the stack.

```
LOCAL_REF0_PUSH
LOCAL_REF1_PUSH
LOCAL_REF2_PUSH
LOCAL_REF3_PUSH
LOCAL_REF4_PUSH
```
The following opcodes are analogous to the local reference ones, but instead of loading values, they store the value of the \texttt{val} register on the local variables

\begin{itemize}
  \item LOCAL\_SET0
  \item LOCAL\_SET1
  \item LOCAL\_SET2
  \item LOCAL\_SET3
  \item LOCAL\_SET4
  \item LOCAL\_SET
\end{itemize}
Chapter 5. Deep variables

Variables which are visible but not in the immediately accessible environment are accessed with the DEEP opcodes.

DEEP_LOCAL_REF
DEEP_LOCAL_SET
DEEP_LOC_REF_PUSH

STklos organizes local environments as this: each level has a maximum of 256 variables. Both the level and the address of local variables are encoded in a single 16-bit integer, as "256v1+v2". For example, 2*256 + 03 = 0x0203. The first byte, 0x02, identifies the level, and the second byte, 0x03, identifies the variable.

The VM will, then, do something like this to access a deep local variable:

```c
/* See this is src/vm.c, CASE(DEEPLOCAL_REF): */
for (level = FIRST_BYTE(info); level; level--)
    e = (SCM) FRAME_NEXT(e);
vm->val = FRAME_LOCAL(e, SECOND_BYTE(info));
```

Here, info is the information to access the variable (a uint16_t number, as every opcode and operand used in the VM). FIRST_BYTE gets the level; SECOND_BYTE gets the var address.

Examples:
In the following example, the value of \(a\) is fetched from a deep environment and pushed onto the stack, so it can be used by the comparison opcode **IN-NUMEQ**:

```lisp
(let ((a 10))
  (lambda ()
    (= a 20)))
```

000: DEEP-LOC-REF-PUSH 256
002: SMALL-INT 20
004: IN-NUMEQ
005: RETURN

The following example shows a variable in a deeper level.

```lisp
(let ((c 4)
      (b 3))
  (lambda ()
    (let ((a 2))
      c)))
```

000: PREPARE-CALL
001: INT-PUSH 2
003: ENTER-TAIL-LET 1
005: DEEP-LOCAL-REF 513
007: RETURN

The number 513 is composed of the bytes 0x02 and 0x01: \#x0201 = 513. This means "the variable of index 1 in level 2" (index 1 is for \(c\), and index 0 is for \(b\)).

The code for `(let ((c 4) (b 3))` is not shown, but it can be seen with **disassemble-exp**:

```lisp
(let ((c 4)
      (b 3))
  (lambda ()
    (let ((a 2))
      c))) #t)
```

000: PREPARE-CALL
001: INT-PUSH 4
003: INT-PUSH 3
005: ENTER-LET 2
007: CREATE-CLOSURE 9 0 ;; ==> 018
010: PREPARE-CALL
011: INT-PUSH 2
013: ENTER-TAIL-LET 1
015: DEEP-LOCAL-REF 513
017: RETURN
018: LEAVE-LET
Chapter 6. Global variables

Global variables can be read and set with the following opcodes:

GLOBAL-REF
GLOBAL-SET

Examples:

```
(disassemble-expr 'my-cool-global-variable) #t)
```

000:  GLOBAL-REF           0

Constants:
0: my-cool-global-variable

```
(disassemble-expr '(set! my-cool-global-variable #f) #t)
```

000:  IM-FALSE
001:  GLOBAL-SET           0

Constants:
0: my-cool-global-variable

6.1. UGLOBAL_{REF,SET} and the checked global table

The instructions GLOBAL_REF and GLOBAL_SET do the following:

1. Acquire the mutex
2. Fetch the index of the global variable
3. Lookup the variable in the current environment (that is, consult a hash table in a module)
4. Verify if the variable is mutable or not
5. Finally, do the real get or set operation
6. Release the lock

Steps 1-4 are quite expensive, and shouldn’t need to be done every time the variable is accessed. Thus, the STklos VM keeps a table with checked globals. The first time a variable is referenced, the VM goes through all those steps, but before releasing the lock there is another step:

5’. Patch the code, changing the GLOBAL_REF or GLOBAL_SET instruction into a UGLOBALREFER or UGLOBAL_SET.
For example, in `GLOBAL_SET` this step is performed by the following two lines:

```c
/* patch the code for optimize next accesses */
vm->pc[-1] = add_global(CDR(ref));
vm->pc[-2] = UGLOBAL_SET;
```

See that what is being changed are the two previous bytecode elements, `pc[-1]` and `pc[-2]`. So the code

```lisp
(set! a 2)
```

would perhaps be translated into

```
000:  SMALL-INT            2
002:  GLOBAL-SET           5
```

where 5 is the index of the variable `a` (as a global).

Then after the first time the `GLOBAL_SET` instruction is performed, the code will **patch itself** and change into

```
000:  SMALL-INT            2
002:  UGLOBAL-SET          n
```

where `n` is the index of this global variable **in a local table**.

The instruction `GLOBAL_SET` takes two integers to be represented, so when `pc[-1]` and `pc[-2]` are changed, what is being changed is the previous argument (5 → n) and the previous instruction (`GLOBAL_SET → UGLOBAL_SET`).

**And**, of course, the `n`-th element of the table contains the address of the variable to be set. This is made clear in the code of `UGLOBAL_SET`:

```c
CASE(UGLOBAL_SET) {
   /* Never produced by compiler */
   /* Because of optimization, we may get re-dispatched to here. */
   RELEASE_POSSIBLE_LOCK;

   fetch_global() = vm->val; NEXT0;
}
```

The checked globals table is defined earlier in `vm.c`:

```c
static SCM** checked_globals;
...```
and the function `add_global(SCM ref)` will add a global to the table.

Of course, this is also done in all other `UGREF_*` instructions in a similar way.

That is why, even using a hash table, access to global variables happens with speed not too far from that of access to local variables in STklos. This can be seen in the following rudimentary benchmark:

```scheme
;;; Using locals: runs in about 3900ms
;;; (let ((a 0)
;;;;       (b 2))
;;;;   (time
;;;;     (repeat 100_000_000
;;;;       (set! a b))))

;;; Using globals: in the same system, runs in about 4000ms
;;; (define a 0)
;;; (define b 2)

;;; (time
;;;;     (repeat 100_000_000
;;;;       (set! a b)))
```
Chapter 7. Operations

7.1. Arithmetic

The operations take the top of stack and \texttt{val} as operands, and leave the result on \texttt{val}.

\begin{verbatim}
IN_ADD2
IN_SUB2
IN_MUL2
IN_DIV2
\end{verbatim}

\begin{verbatim}
(disassemble-expr '(+ a 3) #t)
\end{verbatim}

\begin{verbatim}
000: GLOBAL-REF   0
002: IN-SINT-ADD2 3
\end{verbatim}

Constants:
\begin{verbatim}
0: a
\end{verbatim}

First the value of \texttt{a} (which is the zero-th local variable) is pushed onto the stack. Then, \texttt{DEEP-LOCAL-REF} brings the value of \texttt{x}, and \texttt{IM-ADD2} adds the two values, leaving the result on the local variable register.

For fixnums, the analogous opcodes are:

\begin{verbatim}
IN_FXADD2
IN_FXSUB2
IN_FXMUL2
IN_FXDIV2
\end{verbatim}

\begin{verbatim}
(disassemble-expr '(fx+ v 3))
\end{verbatim}

\begin{verbatim}
000: GLOBAL-REF   0
002: IN-SINT-FXADD2 3
\end{verbatim}

Constants:
\begin{verbatim}
0: v
\end{verbatim}

The following variant of those opcodes do not use the stack. They operate on \texttt{val} and an argument:

\begin{verbatim}
IN_SINT_ADD2
\end{verbatim}
Example:

```
(disassemble-expr '(+ a 2))
```

000: GLOBAL-REF 0
002: IN-SINT-ADD2 2

Constants:
0: a

With `a` as a local variable:

```
(disassemble (lambda (a) (+ a 2)))
```

000: LOCAL-REF0
001: IN-SINT-ADD2 2
003: RETURN

First, the value of `a` is put on `val`; then it is summed with `2`, which comes as an argument to the opcode `IN-SINT-ADD2`.

These also have fixnum variants:

```
IN_SINT_FXADD2
IN_SINT_FXSUB2
IN_SINT_FXMUL2
IN_SINT_FXDIV2
```

Example:

```
(disassemble-expr '(fx+ a 2))
```

000: GLOBAL-REF 0
002: IN-SINT-FXADD2 2

Constants:
0: a
7.2. Increment and decrement val

IN_INCR
IN_DECR

7.3. Comparisons

These compare the top of stack with \texttt{val}, and leave a boolean on \texttt{val}.

\texttt{IN\_NUMEQ} ; \texttt{pop() == val} ?
\texttt{IN\_NUMDIFF} ; ! \texttt{pop() == val} ?
\texttt{IN\_NUMLT} ; \texttt{pop < val} ?
\texttt{IN\_NUMGT} ; \texttt{pop > val} ?
\texttt{IN\_NUMLE} ; \texttt{pop \leq val} ?
\texttt{IN\_NUMGE} ; \texttt{pop \geq val} ?

Example:

\begin{verbatim}
(disassemble-expr '(\geq a 2))
\end{verbatim}

\begin{verbatim}
000: GLOBAL-REF-PUSH 0
002: SMALL-INT 2
004: IN\_NUMGE
\end{verbatim}

Constants:

\begin{verbatim}
0: a
\end{verbatim}

There are also opcodes for \texttt{equal?}, \texttt{eqv?} and \texttt{eq?):

\texttt{IN\_EQUAL}
\texttt{IN\_EQV}
\texttt{IN\_EQ}

Example:

\begin{verbatim}
(disassemble-expr '(eq? a 2))
\end{verbatim}

\begin{verbatim}
000: GLOBAL-REF-PUSH 0
002: SMALL-INT 2
004: IN\_EQ
\end{verbatim}

Constants:
The `dissassemble` procedures will not, however, show the names of symbols or values of strings (`disassemble-expr` does, when passed the extra `#t` argument).

```
(dissassemble (lambda (a) (eq? a 'hello-i-am-a-symbol)))
```

```
000: LOCAL-REF0-PUSH
001: CONSTANT 0
003: IN-EQ
004: RETURN
```

```
(dissassemble-expr '(eq? a 'hello-i-am-a-symbol) #t)
```

```
000: GLOBAL-REF-PUSH 0
002: CONSTANT 1
004: IN-EQ
005:
```

Constants:
0: a
1: hello-i-am-a-symbol

### 7.4. Constructors

These will build structures with the value in `val` and store the structure (that is, the tagged word representing it) again on `val`.

```
IN_CONS
IN_CAR
IN_CDR
IN_LIST
```

Examples:

```
(dissassemble-expr '(cons "a" "b") #t)
```

```
000: CONSTANT-PUSH 0
002: CONSTANT 1
004: IN-CONS
005:
```
### 7.5. Structure references

The following opcodes access and set elements of strings and vectors.

```
IN_VREF
IN_SREF
IN_VSET
IN_SSET
```

V stands for vector, S stands for string; then, REF and SET mean reference '' and set". 

---

 Constants:
  0: "a"
  1: "b"

```
(disassemble (lambda (a b) (cons a b)))
```

```
000: LOCAL-REF1-PUSH
001: LOCAL-REF0
002: IN-CONS
003: RETURN
```

The element to be consed is pushed on the stack; then the second element is loaded on `val`, and then `IN-CONS` is called.

```
(disassemble (lambda (a) (list a)))
```

```
000: LOCAL-REF0-PUSH
001: IN-LIST 1
003: RETURN
```

```
(disassemble-expr (car a) #t)
```

```
000: GLOBAL-REF 0
002: IN-CAR
003:
```

Constants:
  0: a

---

V stands for vector, S stands for string; then, REF and SET mean reference '' and set". 

---

19
The instructions will use the object in the stack and the index from the \texttt{val} register.

Examples

\begin{verbatim}
(let ((a #(0 1 2 3)))
  (lambda () (vector-ref a 2)))
\end{verbatim}

\begin{verbatim}
000: DEEP-LOC-REF-PUSH 256
002: SMALL-INT 2
004: IN-VREF
005: RETURN
\end{verbatim}

In the following example, the \texttt{CONSTANT-PUSH} is including a reference to the string on the stack.

\begin{verbatim}
(disassemble-expr '(string-ref "abcde" 3) #t)
\end{verbatim}

\begin{verbatim}
000: CONSTANT-PUSH 0
002: SMALL-INT 3
004: IN-SREF
005:
\end{verbatim}

Constants:
0: "abcde"

When setting a value, the reference to the vector or string and the index go on the stack (index below the reference to the object – the index is popped first), and the value goes on \texttt{val}, then the setting opcode is used:

\begin{verbatim}
(let ((v (vector #a #b #c)))
  (lambda () (vector-set! v 2 10)))
\end{verbatim}

\begin{verbatim}
000: DEEP-LOC-REF-PUSH 256 ; push ref. to vector
002: INT-PUSH 2 ; push index
004: SMALL-INT 10 ; put new value in val
006: IN-VSET ; set it!
007: RETURN
\end{verbatim}
Chapter 8. Control flow

The following opcodes have an argument, which is the offset to be added to the program counter.

GOTO ; unconditionally jump
JUMP_TRUE ; jump if val is true
JUMP_FALSE ; jump if val is false
JUMP_NUMDIFF ; jump if ! pop() = val (for numbers)
JUMP_NUMEQ ; jump if pop() = val (for numbers)
JUMP_NUMLT ; jump if pop() < val
JUMP_NUMLE ; jump if pop() <= val
JUMP_NUMGT ; jump if pop() > val
JUMP_NUMGE ; jump if pop() >= val
JUMP_NOT_EQ ; jump if pop() not eq? val
JUMP_NOT_EQV ; jump if pop() not eqv? val
JUMP_NOT_EQUAL ; jump if pop() not equal? val

Example:

(disassemble
  (lambda () (if #t 2 4)))

000:  IM-TRUE
001:  JUMP-FALSE  3  ;; ==> 006
003:  SMALL-INT  2
005:  RETURN
006:  SMALL-INT  4
008:  RETURN

STklos’ disassemble is nice enough to tell you the line number where a jump goes!
Chapter 9. Closures, let, and related

9.1. let

The opcodes for ‘entering ‘let’ create new environments and push them on the stack, but do not update activation records, since there is no procedure call happening. Then, the ‘LEAVE_LET opcode removes the environment from the stack.

```
ENTER_LET
ENTER_LET_STAR
ENTER_TAIL_LET
ENTER_TAIL_LET_STAR
LEAVE_LET
```

Examples:

```
(disassemble-expr '(list (let ((x 1))
  x)) #t)
```

```
000:  PREPARE-CALL
001:  ONE-PUSH
002:  ENTER-LET            1
004:  LOCAL-REF0
005:  LEAVE-LET
006:  PUSH
007:  IN-LIST              1
```

Constants:

When the `let` is in tail position, then the opcode used is the ordinary `ENTER_TAIL_LET`, and no `LEAVE_LET` is needed:

```
(disassemble
 (lambda ()
   (let ((x 1))
    x)))
```

```
000:  PREPARE-CALL
001:  INT-PUSH             4
002:  ENTER-TAIL-LET       1
004:  LOCAL-REF0
005:  RETURN
```

Chapter 10. Miscelannea

The following opcode does nothing:

NOP

The following sets the docstring and the formal parameter list documentation for a procedure:

DOCSTRG
FORMALS

Examples:

```
(disassemble-expr (define (f) "A well-documented function" 5) #t)
```

```
000:  CREATE-CLOSURE  4 0  ;; ==> 006
003:  SMALL-INT       5
005:  RETURN
006:  DOCSTRG         0
008:  DEFINE-SYMBOL   1
010:

Constants:
0: "A well-documented function"
1: f
```

```
(disassemble
 (lambda ()
   (define (f) "A well-documented function" 5)
   10))
```

```
000:  PREPARE-CALL
001:  FALSE-PUSH
002:  ENTER-TAIL-LET  1
004:  CREATE-CLOSURE  4 0  ;; ==> 010
007:  SMALL-INT       5
009:  RETURN
010:  DOCSTRG         0
012:  LOCAL-SET0      10
013:  SMALL-INT       10
015:  RETURN
```

Here, DOCSTRG seems to have a zero argument because it uses a constant string, and disassemble does
not show values of strings and symbol names.

The `FORMALS` opcode is similar to `DOCSTRG`, except that it expects a list instead of a string.

```
(compiler:keep-formals #t)

(disassemble-expr '(define (f a b . c)
                   "A well-documented function"
                   (* a 3))
                 #t)
```

```
000:  CREATE-CLOSURE       5 -3;; ==> 007
003:  LOCAL-REF2
004:  IN-SINT-MUL2         3
006:  RETURN
007:  FORMALS              0
009:  DOCSTRG              1
011:  DEFINE-SYMBOL        2
013: 

Constants:
0: (a b . c)
1: "A well-documented function"
2: f
```

### 10.1. Creating closures and procedures

The following opcode creates a closure.

```
CREATE_CLOSURE
```

This opcode fetches two parameters:

- the number of instructions ahead that the VM needs to jump to (because what follows is the code of a closure being created, and it should not be executed, so the VM will jump over it)
- the closure arity.

Examples:

```
(disassemble
 (lambda ()
  (lambda () "Hello")))
```

```
000:  CREATE-CLOSURE       4 0 ;; ==> 006
```
10.2. Procedure calls

The following opcodes are used to make procedure calls:

- **PREPARE-CALL** pushes an activation record on the stack.
- **INVoke** opcodes call procedures – local or global; in tail position or not. The ones with the **PUSH**
prefix also push an argument onto the stack.

These are handled in the VM as states in the state machine (they are labels used in the CASE’s in `vm/.c`).

In `vm.c`, all these instructions end up sending the control to the `FUNCALL` label, which will then check what to do depending on the type of call (`tc_instance`, `tc_closure`, `tc_next_method`, `tc_apply`, or some primitive, `tc_subr…`) 

The peephole optimizer will combine `PUSH`, `GLOBAL-REF INVOKE` instructions, yielding combined instructions. The following is an excerpt from `peephole.stk` where these transformations are documented:

```
;; [GLOBAL-REF, PUSH] => GLOBAL-REF-PUSH
;; [PUSH GLOBAL-REF] => PUSH-GLOBAL-REF
;; [PUSH-GLOBAL-REF, INVOKE] => PUSH-GREF-INVOKE
;; [PUSH-GLOBAL-REF, TAIL-INVOKE] => PUSH-GREF-TAIL-INV
;; [PUSH, PREPARE-CALL] => PUSH-PREPARE-CALL
;; [GLOBAL-REF, INVOKE] => GREF-INVOKE
;; [GLOBAL-REF, INVOKE] => GREF-INVOKE
;; [GLOBAL-REF, TAIL-INVOKE] => GREF-TAIL-INVOKE
;; [LOCAL-REFx, PUSH] => LOCAL-REFx-PUSH
```

The arguments to the `INVOKE`-like instructions are:

- **`INVOKE`**: `n_args` (the procedure address is the first item on the stack, so it is not passed as argument in the code)
- **`GREF-INVOKE`**: `proc_addr, n_args`
- **`PUSH-GREF-INVOKE`**: `first_arg, proc_addr, n_args` (pushes the first and calls the procedure with `n_args` arguments form the stack)

```
(disassemble (lambda () (f)))
```

```
000:  PREPARE-CALL
001:  GREF-TAIL-INVOKE     0 0
004:  RETURN
```

```
(disassemble (lambda () (f 3)))
```

```
000:  PREPARE-CALL
001:  INT-PUSH             3
003:  GREF-TAIL-INVOKE     0 1
006:  RETURN
```
In the next example, GREF-INVOKE is called with arguments 0 and 0. The first value 0 is the address of the procedure in the stack. The IN-SINT-ADD2 procedure is called afterwards to sum 3 with the return from f.

\[
\text{(disassemble (lambda () (+ 3 (f))))}
\]

000: PREPARE-CALL
001: GREF-INVOKE          0 0
004: IN-SINT-ADD2         3
006: RETURN

In the next example, GREF-INVOKE is called with arguments 0 and 2. The value 0 is the address of the procedure in the stack; 2 is the number of arguments given in this procedure call. The IN-SINT-ADD2 procedure is called afterwards to sum 5 with the return from f.

\[
\text{(disassemble (lambda (x) (+ 5 (f x #f)))))}
\]

000: PREPARE-CALL
001: LOCAL-REF0-PUSH
002: FALSE-PUSH
003: GREF-INVOKE          0 2
006: IN-SINT-ADD2         5
008: RETURN

Now the next example shows how INVOKE is used to call a procedure that is non-global (it is in the local environment). The INVOKE instruction will use the first value on the stack as the address of the procedure (it’s DEEP-LOCAL-REF 256, since f is defined inside the let). The other two arguments to be popped from the stack are #f (pushed by the FALSE-PUSH instruction) and the global variable y (pushed by the instruction GLOBAL-REF-PUSH 0). After INVOKE calls f, the instruction IN-SINT-ADD2 3 will sum 3 to the result.

\[
\text{(let ((f (lambda (x) x)))}
\text{(disassemble (lambda () (+ 3 (f y #f))))})
\]

000: PREPARE-CALL
001: GLOBAL-REF-PUSH      0
003: FALSE-PUSH
004: DEEP-LOCAL-REF       256
006: INVOKE               2
008: IN-SINT-ADD2  3
010: RETURN
Chapter 11. Modules

The following opcode enters a given module.

\[ \text{SET\_CUR\_MOD} \]

An SCM object of type `module` must be in the `val` register.

Example:

\[
\text{(disassemble-expr '(select-module m) #t)}
\]

000: PREPARE-CALL
001: CONSTANT-PUSH 0
003: GREF-INVOKE 1 1
006: SET-CUR-MOD
007:

Constants:
0: m
1: find-module

In the above example, the constants were two symbols: `m` and `find-module`. The `find-module` procedure, which is called, will leave module `m` in the `val` register, which is then used by `SET\_CUR\_MOD`.

The following opcode defines a variable in a module.

\[ \text{DEFINE\_SYMBOL} \]

It will define a variable with name set as symbol fetched after the opcode, and value in the `val` register.

\[
\text{(disassemble-expr '(define a "abc") #t)}
\]

000: CONSTANT 0
002: DEFINE-SYMBOL 1
004:

Constants:
0: "abc"
1: a
(disassemble-expr '(define a #f) #t)

000: IM-FALSE
001: DEFINE-SYMBOL 0
003:

Constants:
0: a
Chapter 12. **vm.c**

An important observation:

- **apply**: there is a `DEFINE_PRIMITIVE("apply", ...), but it is **not** used. It is necessary just so there is a primitive of the type `tc_apply`. When the VM finds a primitive of this kind, it'll treat it differently.

Some basic functions in the VM:

- **push(v)**: pushes `v` on the stack (the stack pointer is decreased)
- **pop()**: pops a value from the stack (the stack pointer is increased)
- **fetch_next()** fetches the **next** opcode, increasing the PC
- **fetch_const()** fetches the **next** opcode and uses it as index for a constant
- **look_const()** looks at the current opcode and uses it as index for a constant
- **fetch_global()** fetches the **next** opcode and uses it as index for a global variable
- **add_global(ref)** adds `ref` to the list of global variables, and returns its index. If it was already there, the old index is returned. If it was not, a place is allocated for it, and the new index is returned.

Already covered before:

- **SCM STk_C_apply(SCM func, int nargs, ...)**: applies `func`, with `nargs` arguments
- **SCM STk_C_apply_list(SCM func, SCM l)**: applies `func`, with a list of arguments
- **SCM STk_n_values(int n, ...)**: prepares `n` values in the VM (for the next instruction), and returns a pointer to the `vm+val` register
- **SCM STk_values2vector(SCM obj, SCM vect)**: turns a `values` object into an array with the values

### 12.1. The global lock

There is one global mutex lock for STklos, called `global_lock`, declared in **vm.c**:

```c
MUT_DECL(global_lock); /* the lock to access checked_globals */
```

As per the comment, its purpose is to discipline access to global variables.

Three macros are used to control the global lock (a mutex):

- **LOCK_AND_RESTART** will acquire the lock, and decrease the program counter. It will also set a flag that signals that the lock has been acquired by this thread, and then call **NEXT**. The name “AND_RESTART” reflects the fact that it decreases the PC and calls **NEXT** (for the next instruction) — so the effect is to start again operating on this instruction, but this time with the lock.
- **RELEASE_LOCK** will release the lock, regardless of the thread having it or not. The flag indicating ownership by this thread is cleared.
**RELEASE_POSSIBLE_LOCK** will release the lock **if** this thread has it.

### 12.2. run_vm(vm_thread *vm)

After some initial setup, this function will operate as a state machine. Its basic structure is shown below.

The **CASE** symbol is defined differently, depending on the system, but **CASE(x)** semantically simialar to **case x**: (if computed GOTOs are better, then it’s defined as a label instead — see its definition in **vm.c**).

```c
for (; ; ) {

    byteop = fetch_next(); /* next instruction */

    switch (byteop) {

        CASE(NOP) { NEXT; }
        CASE(IM_FALSE) { vm->val = STk_false; NEXT1; }
        CASE(IM_TRUE) { vm->val = STk_true; NEXT1; }

        ...  

        CASE(PUSH_GLOBAL_REF)
        CASE(GLOBAL_REF) {

        ... (several cases here)

        FUNCALL: /* we "goto" here for procedure invoking from other places in the VM */

        ...  

        STk_panic("abnormal exit from the VM"); /* went through the switch(byteop) */

    }

}  
```
Chapter 13. Continuations

There are undocumented primitives in `vm.c` that can be used to capture and restore continuations. They are listed here with their undocumented Scheme counterparts:

- `STk_make_continuation()` — (%make-continuation)
- `STk_restore_cont(SCM cont, SCM value)` — (%restore-continuation cont value)
- `STk_continuationp(SCM obj)` — (%continuation? obj)
- `STk_fresh_continuationp(SCM obj)` — (%fresh-continuation? obj)

Continuation is a native type (`tc_continuation`). A continuation object (defined in `vm.h`) contains pointers to the C stack, the Scheme stack and several other data.

Capturing a continuation is carried out by the following steps (these are the actual comments in the function `STk_make_continuation`):

1. Determine the size of the C stack and the start address
2. Determine the size of the Scheme stack
3. Allocate a continuation object
4. Save the Scheme stack
5. Save the C stack

Restoring is easier:

1. Restore the Scheme stack
2. Restore the C stack

And, when the C stack is restored, the VM is back to its original state, except for the global variables.