Accessing Values of Constants (Including Text Constants)

Integer and Boolean constants:

Integer and Boolean constants can simply be pushed to the top of the stack whenever we need to use them. Then instructions that utilize them can access them from the stack.
For example :
4 + 5 would lead to a sequence of push 4, push 5, add.
Likewise,
a := 2 would lead to: ADDR LL offset_of _a ; PUSH 2; STORE

Text constants:

As we encounter text constants, we will keep track of them in the symbol table and make note of their positions in the constant area of the stack as well as their size. So when accessing their values, we can then loop through this area of the stack starting at the recorded start position and going down in memory address from there up to the size of the string.

Accessing Values of Scalar Variables

For scalar variables, we store their offsets as well as the lexical level of their enclosing major scope into the symbol table. Given this information, we can get the address of the variable by:
display[LexicalLevel] + offset. The calculation of offset has been described above.

In order to actually access these values, we would need to call ADDR LL ON (where LL is the lexical level and ON is the offset) to push the address onto the stack and then LOAD to actually load the value from that address to the top of the stack.

Accessing Array Elements

Arrays are allocated on the stack as described in the above section. To access the value at an index in the array, we need to first calculate the address of that location in the array. This can be done by calculating the position of the index relative to the start of the array. Therefore for the following example:
A[a .. b] if we want to access the element at A[i], we can calculate its internal offset as i-a
hence for A[-3 .. 2], A[1] = 1-(-3) = 1+4 = 5.
We can then access it as follows:
Note that an expression may be used as the index: Example: A[x + 3].

```
ADDR LL offset_of_A     //puts the base addr of A onto the stack
PUSH index              //(if index is a constant) or evaluate expression that leads to index i.e. A[4+9]
PUSH lowerBound(A)
SUB                     // puts index -  lowerBound(A) on the top of the stack
ADD                     // adds the above value to the address of A.
LOAD                    //load above memory address
```

For 2D arrays the process will be similar, with the exception that the offset into the array can be calculated as follows
A[i, j] in an array A[lb1 .. ub1 , lb2 .. ub2]
offset from A's address for A[i, j] = (i-lb1)*(ub2-lb2+1) +(j-lb2)

So for A[-2..5, -2..3], A[-1,-1] can be indexed as ADDR LL off_A + (-1-(-2))*(3-(-2)+1)+(-1-(-2))
= ADDR LL of_A + 7 (where of_A = offset of A).
which would make sense since
A[-2, -2] would be at ADDR LL of_A,
A[-2, -1] would be at ADDR LL of_A + 1,
A[-2, 0] would be at ADDR LL of_A + 2,
A[-2, 1] would be at ADDR LL of_A + 3,
A[-2, 2] would be at ADDR LL of_A + 4,
A[-2, 3] would be at ADDR LL of_A + 5,
A[-1,-2] would be at ADDR LL of_A + 6,
and therefore A[-1, -1] would be at ADDR LL of_A + 7

Implementing Arithmetic Operators

To implement +, -, *, / we will use the ADD, SUB, MUL and DIV of the machine instructions. These instructions use the top operand on the stack as the right operand in the operation. Hence to do something like 4 / 5 we would need the following:

```
push 4
push 5
DIV
```

At the end of that, the result of ⅘ will be on the top of the stack..

For a more complicated example consider the following

```
var x: integer
var y: integer
var z: integer
var o: integer
x:=5
y:=2
z:=10
```

Assuming that x,y,z,o already have space allocated for them on the stack,
we can then do the following for each variable x,y,z one at a time:

```
PUSH const (where const is the value we're assigning to a variable)
ADDR LL ON
STORE
```

Suppose LL = 0, and offsets for x, y, z, o are Ox, Oy, Oz, Oo respectively.

```
o := (x * y / z) + x - y
then,
ADDR 0 Ox
LOAD
ADDR 0 Oy
LOAD
MUL
ADDR 0 Oz
LOAD
DIV
ADDR 0 Ox
LOAD
ADD
ADDR 0 Oy
LOAD
SUB
ADDR 0 Oo
STORE
```

would lead to the evaluation of o.

Implementing Comparisons Operators

The machine instructions has less than (LT) and an equals(EQ) operators. As with the arithmetic operators, the top most element on the stack is used as the right operand in these operations.

Hence for **a < b**, we can simply do something like
<eval expn a>
<eval expn b>
LT
This will leave the stack with either “MACHINE_TRUE” or “MACHINE_FALSE” on top.

For **a <= b:**
<eval expn b>
<eval expn a>
LT // this does b>a
PUSH 1
SWAP
SUB // 1 - (b>a) i.e. not(b>a)

For **a = b:**
<eval expn a>
<eval epn b>
EQ

For **a not = b:**

Here we can do an =. This will push MACHINE_TRUE or MACHINE_FALSE to the top of the stack. Note that MACHINE_FALSE = 0 and MACHINE_TRUE = 1.
We can then push 1 to the stack and swap so that we now have 1 as the second to top element on the stack and a=b as the top value on the stack. If we then do a SUB we can get the not value.
Proof: if a=b => False => 0 then 1-0 = 1 => True
if a=b => True => 1 then 1-1 = 0 => False.
<eval expn a>
<eval expn b>
EQ
PUSH 1
SWAP
SUB

For **a >= b:**
<eval expn a>
<eval expn b>
LT
PUSH 1
SWAP
SUB

For **a > b:**
<eval expn b>
<eval expn a>
LT

Implementing Boolean Operators

Note: The implementations of labels are addressed elsewhere in this document.

AND
The AND operation will be short-circuited, to do this we will evaluate the first expression, if it is false, we will branch to a label called firstFalse which will put false on the top of the stack and then the rest of the code will follow. If the first value is true, we continue execution, and the final result is what the second expression evaluates to.

Given an expression of the form “e1 ‘and’ e1”, the machine code is:
<evaluate e1>
push firstFalse
BF
<evaluate e2>
push endAND
BR
firstFalse:
push MACHINE_FALSE
endAND:
<rest of code>

OR
Similar to AND we will generate the code to evaluate the first expression, if it is false, we need to check the second expression which will evaluate to our result. If our first expression is true, we we know that the result is true, so we simply branch to the rest of the code, skipping the evaluation of the second expression. (Note: we duplicate the first expression so we don’t have to reevaluate it when we want to return it which may have some side effects.)

Given an expression of the form “e1 ‘or’ e2”, the machine code is:
<evaluate e1>
DUP
push firstFalse
BF
push endOR
BR
firstFalse:
POP
<evaluate e2>
endOR
<rest of code>

NOT

Given an expression of the form “‘not’ expression”, we will push 1 to the stack and then we will evaluate the expression. We will then use our VM’s SUB instruction which will subtract 1 - expressions value, so an expression of 1 becomes 0, and 0 becomes 1.

Example:
not e1 # where e1 is some expression, the machine code is:
PUSH 1
<code to evaluate e1>
SUB