Variables in the main program

Variables in the main program can be stored as follows:

For scalar variables:

As we enter a scope (in this case the main scope) we can simply push “UNDEFINED” for each of the variables in it onto the stack.
Our symbol table will have the offsets stored based on the previous entries in the table within the scope. For any variable in the main program, the offset can be calculated by adding the size of the previous variable to the offset of the previous variable. In the case where the previous variable was scalar, its size will be one word. This will differ for arrays and can be calculated as shown below. Additionally, our symbol table will need to store the lexical level of each scope in order to address into the call-stack. For main, this lexical level is 0. Since main has no arguments or return values attached to it, the first variable can have an offset of 0.

For Arrays:

1D Arrays can take the following notations:

A[3] ← A 1D array with legal entries at A[1], A[2], A[3].
B[-2 .. 1] ← A 1D array with legal entries at B[-2], B[-1], B [0] and B [1]

Calculating the sizes of arrays is fairly trivial and just involves taking the absolute value of (upper_bound - lower_bound). In order to calculate the call-stack address offset of a variable following an array in the scope, we need to add the arrays size to the array’s offset.

For each legal index in the array we must PUSH undefined onto the stack to allocate space for the array.

A similar procedure will occur for 2D arrays. We can calculate the size of a 2D array as follows:
for A[a .. b, c...d] size = (Abs(b-a)+1) * (Abs(d-c)+1). So we would need to do size number of PUSH UNDEFINED instructions.

To calculate array elements addresses in the call-stack, we can consider the first legal index into the array to be stored at offset+0. Then to calculate any subsequent entry’s address, we can just add the index’s position relative to the start of the array. Hence for an index x, into the array C[a .. b], we can find the position of C[x] with the calculation: offset+x-a. Note that for an array of the form A[3], the first legal index is always at 1 not 0.

Variables in Procedures and Functions

Variables in procedures and functions would be handled the same way. When storing the offset of a variable, we need to store space for the return address of the function and we need to store the previous value of the display register before we entered the function. This will be discussed in greater depth later when we discuss the activation record for functions and procedures, but for now, all that we need to know is that a variable in a procedure or function can be stored using PUSH UNDEFINED and its offset that will be stored in the symbol table can be calculated as follows:

offset = 1 word for display address + 1 word for return addr + previous variables offset + previous variables size.
(if no previous variables, previous variables offset + previous variables size is 0)

The total address of the variable then = display[LL] + offset.

Variables in Minor Scopes

For variables in minor scopes we keep the lexical level the same as the current major scope. Their offset is calculated the same way as for their enclosing major scope keeping into account all of the variables that were declared before it in the enclosing scope.

Integer and Boolean Constants

For integer and boolean constants, we can simply push their values onto the stack using the opCode for PUSH and the value of the constant. For boolean constants, MACHINE_TRUE and MACHINE_FALSE will be used.

Text Constants

Text constants can get large, and so instead of individually pushing each character of the constant individually (this would be memory inefficient), we want to put text constants into the “Consts and Misc Data” area of the stack (the area that grows between memorySize-1 and mlp) before runtime. This can be done with the function Machine.writeMemory(short addr , short value).
A text constant can be padded with 8 leading 0’s to make it 16bits and we can then call write memory for each character in the string at locations starting from mlp and decrementing mlp as we go. We can then store the starting address and the length of the text constant into the symbol table so that it can be accessed later.