Activation Record for Functions and Procedures

The activation record holds the arguments, variables etc. for active functions.
Our activation records will look like:

LOCAL VARIABLES
RETURN ADDRESS ← offset 1
DISPLAY ADDRESS ← offset 0
ARGUMENT 1 ← offset -1
…..
ARGUMENT n
RETURN VALUE (if function)

BOTTOM OF ACTIVATION FRAME

As such, the activation frame needs to hold 5 main things:

1. The return value of the function if it is a function and not a procedure
2. The arguments of the function.
3. The display address previously in the display register at the lexical level of the active function.
4. The return address of the function (where in the code to go to after the function terminates).
5. The local variables of the function.

1-4 above are all set in the activation by the caller of the function.
5 is set by the callee.

Procedure and Function Entrance Code

The following explanation applies to functions and procedures with the exception that in the case of procedures we ignore the part about return values. For convenience, I will be referring to functions but this still applies to procedures. When we encounter a function call we need to do the following:

In the Caller:
(Note that return values and arguments can only ever be ints and booleans by the language specification and will therefore always only be one word in length)

- Push space for a return value → PUSH UNDEFINED.

- Push the arguments: For each of the arguments we need to evaluate and push them one at a time starting from the last argument and working towards the first. This will be described in more detail later.

- Next we need to replace the value in the display register at the lexical level of the current function with the address of the activation frame. To do this the previous value in the display register so we can restore it on function termination.
This can be accomplished as follows:
PUSHMT ← To push the current stack pointer.
ADDR LL 0 ← To push the address of the previous activation frame.
SWAP ← So that the previous display address is at the address produced by the PUSHMT Instruction
SETD LL ← This will set the display at the pointer to be what is on top of the stack, which at this point will be the address of where the previous display pointer is currently stored on the stack.
The outcome of this will be discussed further later on.

- We also need to push the return address onto the stack. For this we will need to make use of labels. Suppose we call the label "FUNC_RETURN". Then we'd need to do:
PUSH FUNC_RETURN.
and also on the line of code proceeding the function call, we need to add the label for FUNC_RETURN
More on labels will be discussed at the end of this document.

In the Callee:
- We know how many local variables are in the function. We then just need to **PUSH UNDEFINED** for each of them as is described in the storage of scalar variable

Procedure and Function Exit Code

When procedures and functions exit, we need to clean up the stack by popping variables off it. Before we begin the clean up we will have stored the return value. Based on the structure of our activation frame, the return value is going to be at offset -n+1 from the ADDR LL 0 where n is the number of arguments.
The cleanup will follow a procedure consistent with initialization.

In the Callee:
- We know how many local variables are in the function. Suppose it is 5. We can do **PUSH 5** followed by **POPN** to pop all 5 local variables of the stack.
- The value on the top of our stack is now the return address. So we need to have a **BR** instruction so that we can branch to that return address at which point we'll be back in the caller.

In the Caller:
- At this point the top of the stack will now have the display address of the activation frame at the lexical level of the terminated function before it was replaced by the terminated function. We need to restore this address and this can be done with **SETD**
- Following that, we know how many args were in the function from the symbol table and we can **PUSH n** for n args. and then **POPN** to pop them all off the stack at once.
- This now leaves us with the return value at the top of the stack.

Implementing Parameter Passing

Parameter passing has mostly been described above in the function/procedure entrance code section, but to recap, in the caller, after **PUSH UNDEFINED** for the return value, we will look need to evaluate each of the arguments one by one starting at the last one and working towards the first.

Since our activation frame has DISPLAY ADDR at offset 0 for LL, the arguments are stored such that argument 1 is at **ADDR LL -1** and argument n is at **ADDR LL -n**.

for a function call to function foo:
foo(x+4, 1)
we would do the following:

PUSH UNDEFINED	//Push space for the return value.
PUSH 1	// evaluate argument 2 which is a constant so push it.
ADDR LL Offset_x	//Where LL is the lexical level below the function (if function as at 1, LL = 0).
LOAD	
PUSH 4	// Now x is on the stack followed by 1
ADD	// now x +4 is on top of the stack right above the last argument (i.e. 1).

Hence the top of the stack will have

TOP OF STACK

<evaluated value of x+4>
1
Undefined → Return value
….
_____
BOTTOM OF STACK.

Implementing Function Call and Function Value Return

Again, most of the design for how we will do this has already been discussed above. In terms of implementation, functions will have labels that should be stored in the symbol table. When we see a function call we need to emit the following instructions:

PUSH UNDEFINED //for return value
<eval last arg> //described above
…
<eval first arg>
PUSHMT //To push the current stack pointer.
ADDR LL 0 //To push the address of the previous activation frame. LL is the Lexical level of the called function
SWAP //So that the previous display address is at the address produced by the PUSHMT Instruction
SETD LL //Update the display address.
PUSH RETURN_LABEL //Push the return label.
BR FUNC_LABEL //Branch to the function
<Set the Label for RETURN_LABEL to come return to this point>
//Handle clean up
SETD LL // the display address we need to restore should now be on top of the stack and LL should be the LL of the function that just terminated
PUSH <num arguments>
POPN //pop all the arguments.

Now we are left with the return value on the top of the stack!

Implementing Procedure Call

A procedure call will be implemented in the same way as the function call with the exception that there will be no PUSH UNDEFINED on top for the return value. and when we return from the procedure, there will be no return value at the top of the stack once it has returned.

Display Management Strategy

Again, the general idea for display management has been described above. When we see a function/procedure call, before we break to the function label (so within the caller), but after we evaluate the arguments, we need to push the current msp onto the stack. This can be done using **PUSHMT**. We can then Load the value currently in the display variable at the lexical level of the function to be called (i.e. LL+1 at that point) by doing **ADDR LL+1 0** where LL is the current lexical level. Then we need to do a **SWAP** followed by a **SETD**. This will set the display pointer at the function's lexical level to be pointing to the previous DisplayAddress's location in the function's activation record.
This entire set of instructions will proceed as follows:

<eval args>
PUSHMT
ADDR LL+1 0
SWAP
SETD
<push return address> ….

At clean up time, when we break out of the function and return to the caller, we will have the previous display address at the top of the stack. This can then be restored to the display register by simply using **SETD**.