

# Functions and Procedures Template

KyoKeun Park

November 21, 2019

## 1 Activation Record

The activation record will consist of couple of informations in order for callee to gather all the necessary information to carry out the function/procedure, and for caller to successfully execute and return from the callee.

If it is a function, we will first reserve a space in stack in order to hold to return value. Such memory in the stack allows for callee to store its return value, and it also allows caller to retrieve the return value when the callee is done.

Then, for both function and procedure, we must push the return address to the stack. Such address will be used by the callee to return to the caller when it is done.

Finally, if the function or the procedure contains parameters. It will be included in the activation record as well.

All of the mentioned information must be stored in the activation record in this specific order at all times. For instance, an activation record for **function** `F( m : integer , n : boolean ) : integer` found in “A4-3.488” can be represented as such:

```
1 param : n
2 param : m
3 return address
4 return value
```

Note that in this example, the stack grows up. Hence **param : n** is located on top of the stack. A simpler activation record can be observed with **procedure** `P` from same sample file:

```
1 return address
```

In this case, only the return address is necessary, since it is a procedure (no return value necessary) without any parameters.

## 2 Procedure and Function Entrance Code

When the function or a procedure enters, the activation record needs to be properly set up. Please refer to *Section 1* for more information in regards to the information contained within activation record.

Once the activation record has been set up, we must also set the display register of an index equivalent to the caller’s lexical level. More information on this can be found in *Section 7*. Now, we can jump to the callee, which effectively allows the program to start the execution of called function.

Let us take **function** `F(m : integer , n : boolean) : integer` again as an example. The following instructions must be executed:

```
1 SETD 0
2 PUSH -1
3 PUSHMT
4 ADDR (address of argument for m)
5 ADDR (address of argument for n)
6 ADDR (address of F)
7 BR
```

Note that the initial `PUSH -1` will be a placeholder used for return value. Furthermore, our implementation implies that the caller must set up the stack before calling the callee.

### 3 Procedure and Function Exit Code

Do note that in terms of exit code, procedure and functions differs due to the fact that functions must return a value, while procedures do not. The cleanup which caller must do is similar.

For all functions, we must first load the return value into the address which we have reserved for return value. Then for both functions and procedures, we can branch back to the return address in order to return to the caller function.

Furthermore, we must then remove all of the variables that has been declared within this function and procedure before exiting. In order to figure out how much we need to pop by, we can keep track of amount of memory used by the current function/procedure using the symbol table.

Below is an example of exiting function `function F(m : integer , n : boolean) : integer`. Do note that for procedure, instead of load the return value to `LL + 0x0`, we would instead simply use that value of that address to unconditionally branch back to caller, since we will not be storing the return value for procedures.

```
1 PUSH (return value)
2 ADDR 0 0
3 LOAD
4 POP(total memory used by function/procedure)
5 ADDR 0 1
6 BR
```

### 4 Parameter Passing

Parameter passing is done via loading the values to the parameters onto the stack in left-to-right order. This is done after we have populated return values (if the callee is a function) and return address. We can statically determine the address of the given parameters and we also know the exact location of the parameter since they will always be arranged in left-to-right order.

We have the following set of instructions for `function F(m : integer, n : boolean) : integer` function when the user attempts to use `m`.

```
1 ADDR 0 2
2 LOAD
3 // DO SOMETHING WITH VALUE m
```

After `LOAD`, we will have value of `m` stored on top of stack.

### 5 Function Call and Function Value Return

When a function is called, we must set up the stack for the following:

1. Reserved space for return value
2. Return address to the caller
3. Parameters, if any

Furthermore, we must also set the display register at the index equivalent to our lexical level in order to ensure access to variables outside of the given function.

As mentioned in *Section 3*, when a function finishes its execution, it must return a value and then branch back to the caller. An example can be found in *Section 3* as well.

### 6 Procedure Call

Procedure call is similar to function call with few exceptions. Firstly, since it does not need to return any value, we do not reserve any space in stack for stack value. Following is an example of what procedure call, `procedure Q( m : boolean, n : integer, p : boolean)` from A4-3.488, may look like:

```

1 SETD 0
2 PUSHMT
3 ADDR (address of argument for m)
4 ADDR (address of argument for n)
5 ADDR (address of argument for p)
6 ADDR (address of Q)
7 BR

```

Furthermore, since there are no return value, when the procedure finishes its execution, it does not need to set any values. Following is the cleanup code for same procedure, Q:

```

1 ADDR 0 0
2 BR

```

## 7 Display Management Strategy

Display management is simply done with utilization of given procedure and/or function's lexical level, which can be determined statically during compile time. However, the program must keep track of all of the previous callers for a given procedure/function call. This is done in the case where functions from same lexical level calls each other. The most common example of this will be recursion.

For example, given the caller  $a$ , callee  $b$ , and caller's caller  $c$  with same lexical level,  $x$ , when the callee was called, it must have overwrote  $x$ th display register with address of  $a$ . This means that when callee  $b$  finishes its execution, it must restore the  $x$ th display register with  $c$ 's address.

This can be done by pushing additional work on caller's side. Before setting up activation record, the caller can first push the value of the display register of its lexical level. Then, when callee returns, the caller can first restore the display register to point to correct address.