Functions and Procedures Template

KyoKeun Park

November 22, 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. As well as it needs to store the address which resides in display number of the caller. This allows us to restore the address to the display number in the case of conflict.

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:

```
4 param : n
3 param : m
2 prev address in display 1
1 return address
0 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 prev address in display 1
0 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

Upon entering the procedure or function, it must first set its stack's base address in a respective display register. We will be using this display register and an offset value to refer to any local variables we wish to access. We also know that the display number can be statically determined during compile time, since we can refer to the lexical level of the current function/procedure. This can also be kept track within the symbol table.

Let us take `function F(m : integer , n : boolean) : integer` again as an example. Since `F` has a lexical level of 1, we must execute the following instructions upon entrance:

```
1 PUSHMT
2 SETD 1
```

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 $LL + 0$, 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) : intger` 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 the function is called, 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.

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

```
1 PUSH -1
2 ADDR 1 0
3 PUSH (next instruction that needs to be executed after returning)
4 PUSHMT
5 SETD 1
6 ADDR (address of argument for m)
7 ADDR (address of argument for n)
8 PUSH (address of F)
9 BR
```

When we return from the function call, we must first restore the display number, and also remove the activation record from the stack, with an exception of return value. Again, here is an example with function `F`:

```
1 ADDR 1 2
2 SETD 1
3 POP4
```

Once the above instructions are executed, we will be left with the return value of `F`, which we then can use.

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 return 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 1 1
2 SETD 1
3 POP5
```

Where `POP5` will remove everything in the activation record.

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 xth display register with address of a. This means that when callee b finishes its execution, it must restore the xth display register with c's address.

This can be done by pushing additional work on caller's side. Caller can simply keep an old address from display register of callee's display number. Once we return from callee, we can simply restore the value before continuing with our own instructions.