

## Solutions to Written Assignment 6

1. (a) Here is a tail-recursive function that computes factorial:

```
fact2(n:Int, acc:Int) : Int { if n > 0 then fact2(n-1, n*acc) else acc fi };
```

Here  $\text{fact2}(n, 1) = n!$ . Alternately, if you wanted `fact2` to have only one parameter, you could have named the above function `fact3` and defined `fact2(n:Int) : Int { fact3(n,1) }`;

- (b) If we trace out the execution of `fact` and `fact2` computing  $4!$ , we see that they both require at most five activation records on the stack. Below we show the stack during the evaluation of the last recursive call:

AR for <code>fact</code> (4)
AR for <code>fact</code> (3)
AR for <code>fact</code> (2)
AR for <code>fact</code> (1)
AR for <code>fact</code> (0)

AR for <code>fact2</code> (4, 1)
AR for <code>fact2</code> (3, 4)
AR for <code>fact2</code> (2, 12)
AR for <code>fact2</code> (1, 24)
AR for <code>fact2</code> (0, 24)

For `fact`, before a new activation record is pushed onto the stack we do one subtraction (decrease `n` by one). Just before we remove an activation record from the stack (i.e., just before we return) we do one multiplication (multiply `n*fact(n-1)`). For `fact2`, we do the subtraction and multiplication before each new record is pushed onto the stack. When we remove an activation record from the stack all we do is take the result of the recursive call to `fact2` and return it. Thus the only part of the stack frame for `fact2` we are using after a recursive call is the return address.

We can implement `fact2` more efficiently by reusing the same activation record for a recursive call, rather than pushing a new record on the stack. To recursively call `fact2`, we compute new values for `n` and `a` in temporary space, then to do the function call we replace `n` and `a` on the stack with their new values and restart `fact2`.

On the MIPS architecture, there are at least two ways to re-enter `fact2`. One choice is to retrieve the return address from the stack, store it in `$ra`, pop everything except the parameters off the stack, and then unconditionally jump to `fact2`'s entry point (we don't want to clobber `$ra`).

Another choice is to notice that the correct frame pointer and return address are already on the stack, and so we pop everything up to the return address off the stack, and then unconditionally jump just past `fact2`'s entry point to skip the initial set-up code.

In either case, no new space is required. With our new implementation, `fact2(n)` runs in constant space for any `n`, whereas `fact(n)` requires  $O(n)$  space.

For mutually tail-recursive `f` and `g` we need to be a little more careful than in part (b). In this case, `f` and `g` have the same number of arguments, so we could compile `f` and `g` into a single block of code with two entry points `f` and `g`. Then a function `h` that calls either `f` or `g` pushes two arguments onto the stack, stores the return address in `$ra` (on MIPS) and then jumps to either `f` or `g`. `f` or `g` modifies the arguments appropriately and then jumps to the other function (using one of the two strategies described above), or returns `acc` in the base case.

In general, however, `f` and `g` might have different signatures (e.g., take different numbers of parameters) but still be mutually tail-recursive, so we need a more general strategy.

To implement a general tail call to some function `h` we need to replace the current activation record with a new activation record that is correctly layed out for `h`. As before we compute the arguments to `h` in temporary space. Then we retrieve the return address from the stack (on the MIPS we store it in `$ra`). Finally we overwrite the current activation record with the new parameters, shifting the stack pointer just past the last new parameter, and jump unconditionally to `h`.

2. The program that generates this code is

```
def f(n) = if n=0 then 0 else n+f(n-1)
```

```

3. cgen(for i = e1 to e2 by e3 do e4, nt) =
    cgen(e1, nt)
    sw $a0 -nt*4($fp)
    cgen(e2, nt+1)
    sw $a0 -(nt+1)*4($fp)
    cgen(e3, nt+2)
    sw $a0 -(nt+2)*4($fp)
    li $a0 0
    sw $a0 -(nt+3)*4($fp)
    ld $a0 -nt*4($fp)
loop: ld $t1 -(nt+1)*4($fp)
    bg $a0 $t1 finish
    cgen(e4, nt+4)
    sw $a0 -(nt+3)*4($fp)
    ld $a0 -nt($fp)
    ld $t1 -(nt+2)*4($fp)
    add $a0 $a0 $t1
    sw $a0 -nt*4($fp)
    j loop
finish: ld $a0 -(nt+3)*4($fp)

```

Note that we assume that `nt` starts at 1 (at the beginning of a method) and increases by 1 for each temporary allocated (at the recursive `cgen` calls). In lecture, we started at 4 and incremented by 4 for each temporary allocated, but we do not multiply by 4 when computing the address of each temporary.