

Written Assignment 6

Due March 16, 2006

This assignment asks you to prepare written answers to questions on run-time environment and code generation. Each of the questions has a short answer. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work. Remember that written assignments are to be turned in either at the start of lecture or in the CS164 homework box in 283 Soda by 12:30 PM on the due date.

Please write your name, your account name, your TA's name, and your section time on your homework! We need this information so that we can give you credit for the assignment and so that we can return it to you.

1. Suppose f is a function with a call to g somewhere in the body of f :

```
f(...) {  
  ... g(...) ...  
}
```

We say that this particular call to g is a *tail call* if the call is the last thing f does before returning. For example, consider the following functions for computing positive powers of 2:

```
f(x:Int, acc:Int) : Int { if x > 0 then f(x-1, acc*2) else acc fi };  
g(x:Int) : Int { if x > 0 then 2*g(x-1) else 1 fi };
```

Here $f(x, 1) = g(x) = 2^x$ for $x \geq 0$. The recursive call to f is a tail call, while the recursive call to g is not. A function in which all recursive calls are tail calls is called *tail recursive*.

- (a) Here is a non-tail recursive function for computing factorial:

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

Write a tail recursive function `fact2` that computes the same result. (Hint: Your function will most likely need two arguments, or it may need to invoke a function of two arguments.)

- (b) Recall from lecture that function calls are usually implemented using a stack of activation records. Trace through the execution of `fact` and `fact2` both computing $4!$, writing out the stack of activation records at each step (i.e., draw the tree of activation records). (If you were unable to write a tail-recursive version of `fact`, show functions f and g from above computing 2^4 .) Indicate the amount of computation done before, during, and after each record is created or destroyed. Is there any place where you can see potential for making the execution of the tail-recursive `fact2` more time- or space-efficient than `fact` (without changing `fact2`'s source code)? What could you do?
- (c) Now consider the following pair of functions:

```
f(x:Int, acc:Int) : Int { if x > 0 then g(x-1, acc*2) else acc fi };  
g(x:Int, acc:Int) : Int { if x > 0 then f(x-1, acc*5) else acc fi };
```

In this case, the calls to g and f are all tail calls but they are not immediately recursive. Can you extend your answer to part (b) so that a compiler can use only one or two activation records for a call to f or g ? (Hint: Consider the case when the initial invocation of these functions is via a call to f and the case when the initial invocation is via a call to g .)

2. Consider the following MIPS assembly code program. Using the code generation rules from lecture, what source program produces this code?

```

f_entry:
    move    $fp $sp
    sw     $ra 0($sp)
    addiu  $sp $sp -4
    lw     $a0 4($fp)
    sw     $a0 0($sp)
    addiu  $sp $sp -4
    li     $a0 0
    lw     $t1 4($sp)
    addiu  $sp $sp 4
    beq   $a0 $t1 true_branch
false_branch:
    lw     $a0 4($fp)
    sw     $a0 0($sp)
    addiu  $sp $sp -4
    sw     $fp 0($sp)
    addiu  $sp $sp -4
    lw     $a0 4($fp)
    sw     $a0 0($sp)
    addiu  $sp $sp -4
    li     $a0 1
    lw     $t1 4($sp)
    sub    $a0 $t1 $a0
    addiu  $sp $sp 4
    sw     $a0 0($sp)
    addiu  $sp $sp -4
    jal   f_entry
    lw     $t1 4($sp)
    add    $a0 $a0 $t1
    addiu  $sp $sp 4
    b     end_if
true_branch:
    li     $a0 0
end_if:
    lw     $ra 4($sp)
    addiu  $sp $sp 12
    lw     $fp 0($sp)
    jr    $ra

```

3. Give a recursive definition of the cgen function (as given in slide 52 in the lecture notes) for the following new construct.

$$\text{for } i = e_1 \text{ to } e_2 \text{ by } e_3 \text{ do } e_4$$

Assume that the subexpressions e_1, e_2, e_3 and e_4 are integer-valued. A “for loop” expression is evaluated according to the following rules. The first three subexpressions are evaluated once at the start of the loop in the order e_1, e_2 , and then e_3 . The subexpression e_4 is evaluated once per iteration of the loop. The index variable i is initialized to the value of e_1 . The loop bound is the value of e_2 and i is incremented by the value of e_3 after each iteration. The loop terminates before executing an iteration where the value

of i is greater than the loop bound. The value returned by the “for loop” expression is the value of the expression e_4 in the last iteration. If the loop does not execute at all, then the value returned is the integer 0.

Following is a more formal semantics of the for expression in terms of the Cool expressions.

```
let t: Int ←  $e_1$  in
let bound: Int ←  $e_2$  in
let incr: Int ←  $e_3$  in
let result: Int ← 0 in
let i: Int ←  $t$  in
  while ( $i \leq$  bound) loop {
    result ←  $e_4$ ;
     $i \leftarrow i +$  incr;
  } pool;
result
```

Note that the expressions e_1 , e_2 and e_3 are evaluated ONLY once before the start of the loop. Also note that any occurrences of variable i in e_1 , e_2 and e_3 refer to the value of i just before the for loop. Any occurrence of variable i in expression e_4 refers to the loop index variable i .