Subroutines & The Runtime Stack

CS 350: Computer Organization & Assembler Language Programming
Lab 7, due Fri Apr 10

A. Why?

- Information for a procedure call is stored in an activation frame. At runtime, the activation frames form a stack (in C and C++) or heap (in Java).

B. Outcomes

After this lab, you should be able to:

- Describe the contents of the runtime stack and its activation records as routines are called and returned from.

C. Notation

If we label locations in the program and activation frames, we can write out the activation stack without using arrows; here's a modified version of the program from Lecture 16, along with its stack. (See the notes on the next page.)

```
main() {
  int x = 2;
  Sub1(x+1) /* Location 1 */;
}

int Sub1(int x) {
  int w = x+2;
  call Sub2(w,10)
  /* Location 2 */
  set w to result of Sub2
  set RV = w
  return;
}

int Sub2(int q, int r) {
  int k; int m[2];
  /* Location 3 */
  set RV = k;
  return;
}
```

```
Frame for Sub2:
    m[0] = 0
    m[1] = 0
    Location 4: k = 0
    DL = Location 5
    RA = Location 2
    RV = 0
    q = 5
    r = 10
For Sub1:
    Location 5: w = 5
    DL = Location 6
    RA = Location 1
    RV = 0
    x = 3
For main:
    Location 6: x = 2
    DL = null
    RA to OS
    RV = 0
```


Notes:

- In the frames, `DL` means *dynamic link* (to the frame of the caller); `RA` means *return address* (to the location to jump back to in the caller's code); `RV` means *return value* (let's assume it's initialized to 0).
- The frames contain definitions of locations 4, 5, and 6, but location 4 (the top frame of the stack) isn't used.
- In the code, I've broken up statements like `w = Sub2(w,10);` into two parts: call `Sub2(w,10)` and `set w to result of Sub2`. This way, we can be more precise about specifying locations: The location between the two parts is exactly the return address for the call of `Sub2`.
- Similarly, I've broken up `return` *expr*`;` into `RV =` *expr*`;` and `return;`. This way, we can talk about the point in time between copying the return value onto the stack and starting to pop off the frame for the current call.

D. Problems [100 points total]

Problem 1 [50 points]

For the program below, show what the runtime stack looks like whenever execution is at locations 1, 3, or 5.

```
int f(int n) {
    int r = 1;
    if (n <= 1) {
        RV = 1 /* Location 1 */;
        return;
    }
    else {
        call f(n-1) /* Location 2 */;
        set r to result of f
        RV = r*n; /* Location 3 */;
        return;
    }
}
int main() {
    int m = 0;
    call f(3) /* Location 4 */
    set m to result of f; /* Location 5 */
    return 0;
}
```


Problem 2 [50 points]

For the program below, show what the runtime stack looks like whenever execution is at locations 1, 3, or 5.

```
int g(int n, int r) {
    if (n <= 1)
        RV = r;
        /* Location 1: */ return;
    else {
        call g(n-1, r*n) /* Location 2 */
        set RV = result of g
        /* Location 3 */
        return;
    }
}

int main() {
    int m = 0;
    call g(3, 1) /* Location 4 */;
    set m = result of g
    /* Location 5 */
    return;
}
```

Programming Language Note[*]: This is the "tail recursive" version of factorial; the innermost recursive call of `g` sets the common return value used by all the calls of `g`, and every recursive return from `g` leads immediately to another return from `g`, without accessing any parameters or local variables. The upshot is that only the top-level call of `g` actually needs a fresh activation frame to be pushed onto the stack; all the other calls can just reuse the space for the top-level call. Tail-recursive functions can be implemented using loops, which makes them much more efficient to use than non-tail-recursive functions.

[*] This will come up in other courses; you don't need to know it for CS 350.

