Slides by Andrew DeOrio,
Jeff Ringenberg and
Brian Noble

EECS 280

Programming and Introductory Data Structures

Tail Recursion

Recursion

- Recall the recursive factorial function

```
int factorial (int n) {
  // REQUIRES: n >= 0
  // EFFECTS:  computes n!

  if (n == 0) return 1;     // base case
  return n*factorial(n-1); // recursive step
}
```

Another kind of factorial

Re-write the recursive version to use the same amount of space as is required by the iterative version (approximately).

```
int fact_helper(int n, int result) {
  // REQUIRES: n >= 0
  // EFFECTS: returns result * n!

  if (n == 0) return result;
  return fact_helper(n-1, result * n);
}

int factorial(int num) {
  // REQUIRES: num >= 0
  // EFFECTS: returns num!
  return fact_helper(num, 1);
}
```

```
int fact_helper(int n, int result) {
  if (n == 0) return result;
  return fact_helper(n-1, result * n);
}
```

```
// EXAMPLE: factorial(3)
```

Another kind of factorial

- This function is equivalent to the original factorial.
- [redacted]
- There are two steps. First, prove the base case, and second, the inductive step.

```
int fact_helper(int n, int result) {
  // REQUIRES: n >= 0
  // EFFECTS: returns result * n!

  if (n == 0) return result;
  return fact_helper(n-1, result * n);
}

int factorial(int num) {
  // REQUIRES: num >= 0
  // EFFECTS: returns num!

  return fact_helper(num, 1);
}
```

Another kind of factorial

- There is an important thing to notice about fact_helper.

- For **every** call to fact_helper:

```
n! * result == num!
```

- For the first call, this is easy to see, since:

```
n == num
result == 1
```

```
int fact_helper(int n, int result) {
  // REQUIRES: n >= 0
  // EFFECTS: returns result * n!

  if (n == 0) return result;
  return fact_helper(n-1,result*n);
}

int factorial(int num) {
  // REQUIRES: num >= 0
  // EFFECTS: returns num!

    return fact_helper(num, 1);
}
```

Another kind of factorial

- For **every** call to fact_helper:

```
n! * result == num!
```

- For the second call:

```
n == (num - 1)
result == (1*num)
       == num
```

Substituting, we get:

```
(num-1)! * num == num!
```

- This is true by inspection. You can continue unwinding if you like.

```
int fact_helper(int n, int result) {
  // REQUIRES: n >= 0
  // EFFECTS: returns result * n!

  if (n == 0) return result;
  return fact_helper(n-1,result*n);
}

int factorial(int num) {
  // REQUIRES: num >= 0
  // EFFECTS: returns num!

  return fact_helper(num, 1);
}
```

Another kind of factorial

- For **every** call to fact_helper:

```
n! * result == num!
```

- This is called the “recursive invariant” of fact_helper and is something that is always true.

- Being able to write down invariants makes it much easier to write these sorts of functions.

Another kind of factorial

- So what's the big deal?
- This just looks like a more complicated way to write the solution.
- Let's trace out a call to the “new” factorial(2), and compare it to the “old” factorial(2):

```
"new"
factorial(2)
-->fact_helper(2, 1)
   -->fact_helper(1, 2)
      -->fact_helper(0, 2)
      <--2
   <--2
<--2
```

```
"old"
factorial(2)
-->2 * factorial(1)
  -->1 * factorial(0)
    <--1
  <--1*1 (==1)
<--2*1 (==2)
```

Another kind of factorial

- As the two recursions progress, they look the same.
- However, as they “unwind”, the “new” one doesn't do any more work.
- The “new” one simply passes the value from the deepest call out to the top. But, the “old” one still has work to do.

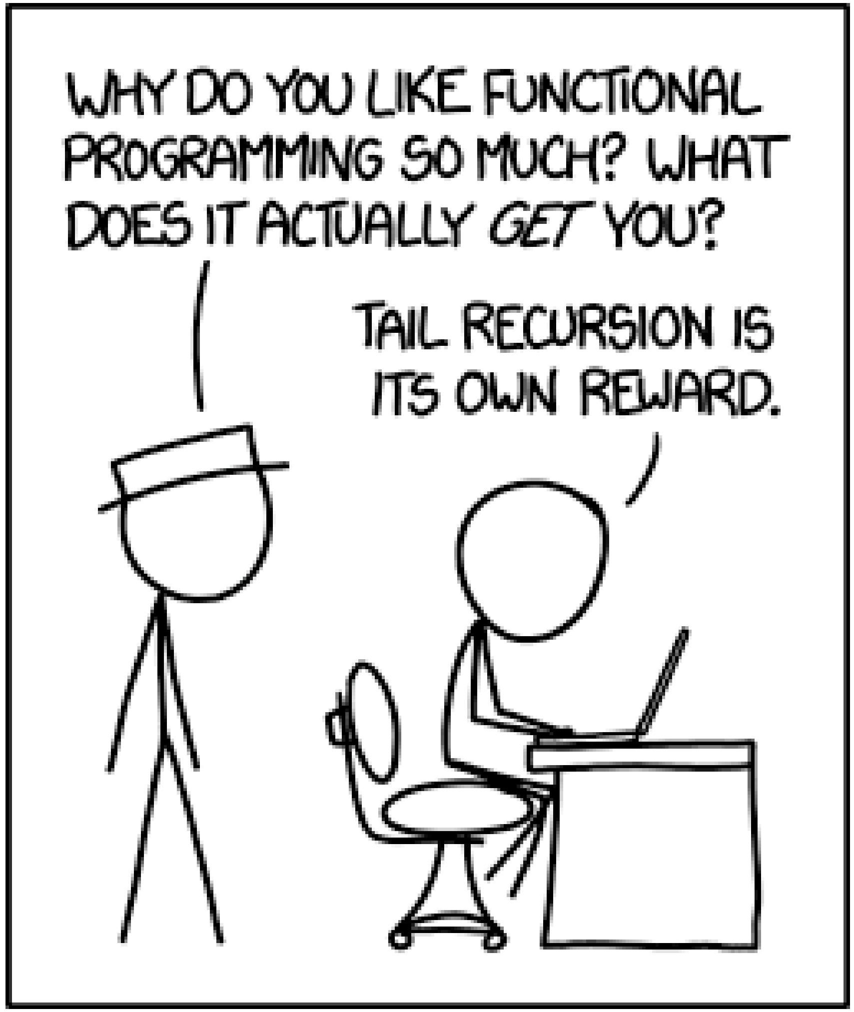
```
"new"
factorial(2)
-->fact_helper(2, 1)
   -->fact_helper(1, 2)
      -->fact_helper(0, 2)
      <--2
   <--2
<--2
```

```
"old"
factorial(2)
-->2 * factorial(1)
  -->1 * factorial(0)
    <--1
  <--1*1 (==1)
<--2*1 (==2)
```

Stack effects

- So what are the effects of this new version on the stack?
- The activation record of a function is needed only as long as there is computation left over and can be discarded as soon as the return value is known.
- With the new version, the concrete return value isn't known at the time of the recursive call. However, we do know that whatever that recursive call returns, that will be our return value too.
- This means that the caller's stack frame isn't needed any more, and we can throw it away.
- This is called **tail recursion**

Tail Recursion



Cartoon: xkcd.com

Tail Recursion

- With tail recursion, there is no pending computation at each recursive step, so we can **re-use** the activation record rather than create a new one.
- Here’s how it works:

```
factorial
 num: 3
```

```
int fact_helper(int n, int result) {
  // REQUIRES: n >= 0
  // EFFECTS: returns result * n!

  if (n == 0) return result;
  return fact_helper(n-1,result*n);
}

int factorial(int num) {
  // REQUIRES: n >= 0
  // EFFECTS: returns num!

    return fact_helper(num, 1);
}
```

Tail Recursion

- factorial calls fact_helper:

```
factorial
 num: 3
```

```
fact_helper
 n:      3
 result: 1
```

```
int fact_helper(int n, int result) {
  // REQUIRES: n >= 0
  // EFFECTS: returns result * n!

  if (n == 0) return result;
  return fact_helper(n-1,result*n);
}

int factorial(int num) {
  // REQUIRES: n >= 0
  // EFFECTS: returns num!

    return fact_helper(num, 1);
}
```

Tail Recursion

- n is not zero, so the alternative is evaluated:

```
return fact_helper(2, 3)
```

```
factorial
 num: [3]
```

```
fact_helper
 n:      [3]
 result: [1]
```

  - This is a tail-recursive call: fact_helper is calling itself, and
    there is no work upon return.

```
int fact_helper(int n, int result) {
  // REQUIRES: n >= 0
  // EFFECTS: returns result * n!

  if (n == 0) return result;
  return fact_helper(n-1,result*n);
}

int factorial(int num) {
  // REQUIRES: n >= 0
  // EFFECTS: returns num!

    return fact_helper(num, 1);
}
```

Tail Recursion

- So, we can **re-use** the storage of the stack frame!

```
factorial
 num: [3]
```

```
fact_helper
 n:      [2]
 result: [3]
```

```
int fact_helper(int n, int result) {
  // REQUIRES: n >= 0
  // EFFECTS: returns result * n!

  if (n == 0) return result;
  return fact_helper(n-1,result*n);
}

int factorial(int num) {
  // REQUIRES: n >= 0
  // EFFECTS: returns num!

    return fact_helper(num, 1);
}
```

`fact_helper(2, 3)`

Tail Recursion

- Same thing:

```
return fact_helper(1, 6)
```

```
factorial
 num: 3
```

```
fact_helper
 n:      1
 result: 6
```

```
int fact_helper(int n, int result) {
  // REQUIRES: n >= 0
  // EFFECTS: returns result * n!

  if (n == 0) return result;
  return fact_helper(n-1,result*n);
}

int factorial(int num) {
  // REQUIRES: n >= 0
  // EFFECTS: returns num!

    return fact_helper(num, 1);
}
```

Tail Recursion

- And again:

```
return fact_helper(0, 6)
```

```
factorial
 num: 3
```

```
fact_helper
 n:      0
 result: 6
```

- Now the result is returned directly to factorial!

```
int fact_helper(int n, int result) {
  // REQUIRES: n >= 0
  // EFFECTS: returns result * n!

  if (n == 0) return result;
  return fact_helper(n-1,result*n);
}

int factorial(int num) {
  // REQUIRES: n >= 0
  // EFFECTS: returns num!

    return fact_helper(num, 1);
}
```

Tail vs. Plain Recursion

- If the result of the recursive call is returned directly with no pending computation, it is tail-recursive. Otherwise, it's "plain" recursion.
- Sometimes, it's easiest to write a recursive function "tail-recursively". When it isn't (as factorial is not), you typically have to invent a helper function to make it all work out.
- Writing a tail recursive version of a function often requires you to add an "extra" argument or two that keeps track of the current "state" of the computation.
- In `fact_helper()`, this extra argument is `result` and it's similar to the local variable in the iterative version. That's no accident!

Tail-recursive or not? Why?

```
void countdown1(int n) {
  if (n <= 0) {
    return;
  } else {
    cout << n << endl;
    countdown1(n-1);
    return;
  }
}
```

```
$ ./a.out
3
2
1
```

Tail-recursive or not? Why?

```
void countdown1(int n) {
  if (n <= 0) {
    return;
  } else {
    cout << n << endl;
    countdown1(n-1);
    return;
  }
} // for comparison
```

```
void countdown2(int n) {
  if (n <= 0) return;
  cout << n << endl;
  countdown2(n-1);
}
```

```
void countdown3(int n) {
  if (n > 0) {
    cout << n << endl;
    countdown3(n-1);
  }
}
```

Bonus question: identify the base case and recursive steps.

Tail-recursive or not? Why?

```
void countdown4_help
(int n, int i) {

  if (i > n) return;
  countdown4_help(n, i+1);
  cout << i << endl;
}

void countdown4(int n) {
  countdown4_help(n, 1);
}
```

Tail-recursive or not? Why?

```
void countdown5(int n) {
  if (n <= 0) return;
  cout << n << endl;
  if (n % 2) { // n is odd
    countdown5(n-1);
    return;
  } else {     // n is even
    cout << n-1 << endl;
    countdown5(n-2);
    return;
  }
}
```