Tail Recursion | Coursera

coursera.org/learn/pointers-arrays-recursion/supplement/7blTK/tail-recursion

Tail Recursion

Tail Recursion

The recursive functions we have seen so far use *head recursion*—they perform computation after they make a recursive call. In the factorial example, after the recursive call to factorial, the returned result is multiplied by *n* before that answer is returned. There is another form of recursion, called *tail recursion*. In tail recursion, the recursive call is the **last** thing the function does before returning. That is, for a tail recursive function f, the only recursive call will be found in the context of **return f (...);** —as soon as the recursive call to f returns, this function immediately returns its result without any further computation.

A generalization of this idea (separate from recursion) is a *tail call*—a function call is a tail call if the caller returns immediately after the called function returns, without further computation. Put another way: if function *f* calls function g, then the call to *g* is a tail call if the only thing *f* does after *g* returns is immediately return *g*'s return value (or if *f*'s return type is **void**, to just return). These two concepts tie together in that a recursive function is tail recursive if-and-only-if its recursive call is a tail call. To understand this definition, let us look at a tail recursive implementation of the factorial function:

```
13

10

11

12

7

8

9

5

6

1

2

3
```

```
4
}

int factorial (int n) {
  return factorial_helper (n, 1); //tail call
  //after recursive call returns, just return its answer
  return factorial_helper (n - 1, ans * n);
}
  }
  //recursive call is a tail call
int factorial_helper (int n, int ans) {
  //base case
  if (n <= 0) {
    return ans;
```

Here we have a tail recursive *helper function*—a function that our primary function calls to do much of the work. Helper functions are quite common with tail recursion, but may appear in other contexts as well. The helper function takes an additional parameter for the answer that it builds up as it recurses. The primary function just calls this function passing in *n* and 1. Notice how the recursive call is a tail call. All that happens after that call returns is that its result is returned immediately. Note that the tail recursive version of factorial does no less work than the original recursive version. In the original recursive version, the factorial of a number is created by a series of multiplications that occur *after* each recursive call. In the tail recursive version, this multiplication takes place *prior* to the recursive call. The running product is stored in the second parameter to the tail recursive function.

The next video will walk you through how tail recursive code is executed. We will first step through this code using all the rules that you are familiar with. Next, we will show you an optimization that many compilers will do to make tail recursive functions more efficient. When a compiler performs "tail recursion elimination" it reuses the current frame for the tail recursive call. Because the function returns immediately after the tail call completes, the compiler recognizes that the values in the frame will never be used again, so it can be overwritten.