CS125 : Introduction to Computer Science

Lecture Notes #29
Accumulator Recursion

Lecture 29 : Accumulator Recursion

Recall that our factorial method was forward-recursive:

```
// Implementation #1 of the factorial algorithm
// This is the algorithm in forward-recursion form
public static int fac(int n)  // n >= 0
{
   if (n > 0)
      return n * fac(n - 1);
   else  // n == 0
      return 1;
}
```

We can't just convert this to a loop and say `n = n - 1` in place of the recursive call, because we still need that original value of `n` to do the multiplication once we return from the recursive call. So making a recursive function of this kind into a loop is not the simple process that it was with a tail-recursive method. On the other hand, we know a loop-based factorial method is possible, because we wrote one back in Lecture Notes #21. So how did that loop-based method come about? How could we write a loop-based method if our factorial algorithm above cannot be converted into a loop?

The link between the two is a type of recursion is known as *accumulator recursion.* Accumulator recursion is a form of tail recursion; the recursive call will be the final work in the recursive case. However, it's tail recursion in which we have to add extra "unnecessary" parameters in order to be able to write it in tail-recursive form. If we wrote it in forward-recursive form, these parameters would not be needed, but in order to write it in tail-recursive form, we need to add extra parameters that "accumulate" a result in some way so that we don't need to do work as we return from the recursive calls. That's why it is known as accumulator recursion.

In the case of the factorial method, the only parameter we need for the calculation is the value of `n`; as proof of this, we can simply look at the code above, which only has `n` as a parameter and yet works perfectly! However, if we want to write a tail-recursive method for calculating factorial, we will need to add an *additional* parameter that was *not* needed for the forward-recursive version. Specifically, what we want to do is to add in a parameter that will accumulate the product of the different values of `n`; we will thus add a second integer parameter to the factorial method. If we have that second parameter:

```
   public static int fac(int n, int productSoFar)
```

then our recursive case could multiply `n` by `productSoFar` *before* the recursive call is made, and thus pass that value as an argument:

```
  return n * fac(n - 1)     -------> return fac(n - 1, n * productSoFar)

  (old recursive case)               (new recursive case)
```

That gives us the following code, which is the start of a new factorial method:

```
public static int fac(int n, int productSoFar)  // n >= 0, productSoFar >= 1
{
   if (n > 0)
      return fac(n - 1, n * productSoFar);
   // still need base case
}
```

What we are doing here is doing our multiplications *before* we make the recursive call...but it does require passing an extra parameter to hold the product so far. Then, when we reach the base case, we have the entire factorial product and can just return that:

```
// Implementation #2 of the factorial algorithm
// This is the algorithm in accumulator-recursion form
// The first call to this method should send the value 1 as the second argument
public static int fac(int n, int productSoFar)  // n >= 0, productSoFar >= 1
{
   if (n > 0)
      return fac(n - 1, n * productSoFar);
   else  // n == 0
      return productSoFar;
}
```

In order to correctly begin the recursion, we need to send an initial value to `productSoFar`, as well as sending the initial value we want the factorial of, to `n`. Generally, we would initialize a product variable to `1` (just as we would generally initialize a `sum` variable to `0`), so to calculate the factorial of `k`, we would no longer call `fac(k)`, but rather, `fac(k, 1)`.

We can see the difference between the two, on the table on the next page. For both examples, we assume all the method calls down to the base case have been made, but that the base case has not been returned from yet. On the left, we see the method notecards for the forward-recursive version; on the right, we see the method notecards for the accumulator-recursive version.

```
Implementation #1                        Implementation #2
--------------------                    --------------------------

-----------------                       ------------------------
| main                                  | main
|   int x;                              |    int x;
|   x = fac(4);                         |    x = fac(4, 1);
|                                       |
|                                       |
|________________                       |_________________________
| fac     (n: 4)                        | fac         (n: 4)
|  (#1)                                 |  (#1)       (productSoFar: 1)
|                                       |
|  need: fac(3)                         | need: fac(3, 4)
|__________________                     |_________________________
| fac     (n: 3)                        | fac         (n: 3)
|  (#2)                                 |  (#2)       (productSoFar: 4)
|                                       |
|  need: fac(2)                         | need: fac(2, 12)
|____________                           |__________________________
| fac     (n: 2)                        | fac         (n: 2)
|  (#3)                                 |  (#3)       (productSoFar: 12)
|                                       |
|  need: fac(1)                         | need: fac(1, 24)
|____________                           |__________________________
| fac     (n: 1)                        | fac         (n: 1)
|  (#4)                                 |  (#4)       (product: 24)
|                                       |
|  need: fac(0)                         | need: fac(0, 24)
|____________                           |__________________________
| fac     (n: 0)                        | fac         (n: 0)
|   (#5)                                |  (#5)       (product: 24)
|                                       |
|  base case!                           |  base case!
|____________                           |___________________________
```

As we return from the various method calls, from #5 to #4 to #3 to #2 to #1, the forward-recursive version needs to do some work – multiplications – before each return (except for the return from the base case), but for the accumulator-recursive method, all the work is already done, and we just keep returning the same value:

```
Implementation #1
--------------------

-----------------
| main
|   int x;
|   x = fac(4);
|
|         [24 written into x]
|_________________
| fac     (n: 4)
|  (#1)
|           ----> returns 24
|  need: fac(3)
|__________________
| fac     (n: 3)
|  (#2)
|           ----> returns 6
|  need: fac(2)
|____________
| fac     (n: 2)
|  (#3)
|           ----> returns 2
|  need: fac(1)
|____________
| fac     (n: 1)
|  (#4)
|           ----> returns 1
|  need: fac(0)
|____________
| fac     (n: 0)
|   (#5)
|           ----> returns 1
|  base case!
|____________
```

```
 Implementation #2
--------------------------

------------------------
| main
|    int x;
|    x = fac(4, 1);
|
|              [24 written into x]
|__________________________
| fac         (n: 4)
|  (#1)       (productSoFar: 1)
|
| need: fac(3, 4)  ----> returns 24
|__________________________
| fac         (n: 3)
|  (#2)       (productSoFar: 4)
|
| need: fac(2, 12)  ----> returns 24
|___________________________
| fac         (n: 2)
|  (#3)       (productSoFar: 12)
|
| need: fac(1, 24)  ----> returns 24
|___________________________
| fac         (n: 1)
|  (#4)       (product: 24)
|
| need: fac(0, 24)  ----> retursn 24
|___________________________
| fac         (n: 0)
|  (#5)       (product: 24)
|
|  base case!       ----> returns 24
|____________________________
```

Each recursive call in the accumulator-recursive version will result in the value of the argument `n - 1` being written back into the parameter `n` of the next call, and the value of the argument `n * productSoFar` begin written into the parameter `productSoFar` of the next call.

Since accumulator recursion is just a form of tail recursion, that means that accumulator-recursive algorithms can be converted into loops using the technique we already discussed:

```
public static int fac(int n, int productSoFar)  // n >= 0, productSoFar >= 1
{
   while (n > 0)   // while our recursive-case condition is still met
   {
      // nothing was done in the recursive case, before the recursive call
      // then, we have params = args;
      n = n - 1;
      productSoFar = n * productSoFar;
   }
   return productSoFar;   // this was originally the base case code
}
```

We actually have a slight problem here; we are changing `n`, and then using `n` in the expression `n * productSoFar`. Unfortunately, that multiplication expression expects the *old* value of `n`, not the new one. When we had the line in recursive-call form:

```
   fac(n - 1, n * productSoFar)
```

then both arguments were evaluated, using the old value of `n`. But now that we have two assignment statements, rather than the passing of two arguments to two parameters, the effects of the earlier assignments exist when we try to perform the later assignments. To get around this, we either need a temporary variable:

```
public static int fac(int n, int productSoFar)  // n >= 0, productSoFar >= 1
{
   while (n > 0)   // while our recursive-case condition is still met
   {
      // nothing was done before the recursive call
      // the, we have params = args;
      int tempNewN = n - 1;
      productSoFar = n * productSoFar;
      n = tempNewN;  // now we can safely store the new value into n, since
                     // we are through using the old value of n
   }
   return productSoFar;   // this was originally the base case code
}
```

or else we can simply reorder those two assignments, which is okay since the evaluation of the new value of `n`, does NOT rely on the old value of `productSoFar`, unlike how the evaluation of the new value of `productSoFar` relies on the old value of `n`. So, the code we get in that case is as follows:

```
public static int fac(int n, int productSoFar)  // n >= 0, productSoFar >= 1
{
   while (n > 0)   // while our recursive-case condition is still met
   {
      // nothing was done before the recursive call
      // the, we have params = args;
      productSoFar = n * productSoFar;
      n = n - 1;
   }
   return productSoFar;   // this was originally the base case code
}
```

Finally, for the loop version, we don't actually need to pass in a "`productSoFar`" value. Rather than requiring that the client send in a `1` as an argument to this parameter, we can simply make it a local variable, and initialize it to `1` ourselves.

```
// Implementation #3 of the factorial algorithm
// This is the algorithm in loop-based form
public static int fac(int n)  // n >= 0
{
   int productSoFar = 1;
   while (n > 0)
   {
      productSoFar = n * productSoFar;
      n = n - 1;
   }
   return productSoFar;
}
```

The above code is basically the same code you saw in Lecture Notes #21 when we first presented the loop-based version of factorial! Note that this means we *cannot* write the loop-based version, without the use of the extra variable `productSoFar`. In order to write the algorithm in tail-recursive (accumulator-recursive) form, you *needed* the extra variable (as a parameter, in that case); without it, you could only write the algorithm in forward-recursive form. And the conversion from tail-recursion to a loop-based version, cannot get rid of needed work like continually writing into the `productSoFar` variable. Converting to a loop merely reorganizes when and where certain computations get done, to make additional method notecards unnecessary; it can't *eliminate* the need for those computations, or for the variables that are used in those computations. So, while we were able to make the `productSoFar` variable a local variable rather than a parameter variable – a change of organization, not computation – we cannot eliminate that variable entirely.

That is the connection between our forward-recursive definition of factorial, and our loop version of factorial. We add the extra variable `productSoFar` into our computation, and now that we have access to that extra variable to use as an accumulator, we can convert from forward-recursion to accumulator-recursion, and then to a loop-based method. Without that extra variable, we are forced to perform our computation in a forward-recursive way, so that we can make use of the saved values in different method notecards, to aid us in our computation.

Another example would be our exponentiation algorithm:

```
// Implementation #1 of the exponentiation algorithm
// This is the algorithm in forward-recursive form
public static int pow(int base, int exp)
{
   if (exp == 0)
      return 1;
   else  //  exp > 0
      return base * pow(base, exp - 1);
}
```

After we finish a recursive call, there is more work to do besides just returning a value – namely, the multiplicaton of `base` and our recursive call result. So, this is a forward-recursive algorithm. We can try and convert this to an accumulator recursion implementation using the same technique we used for the factorial computation – instead of multiplying the `base` by our recursive result, we could try multiplying the base by an existing product, and then passing that product as a third parameter to our exponentiation method:

```
// Implementation #2 of the exponentiation algorithm
// This is the algorithm in accumulator-recursive form
// The first call to this method should send the value 1 as the third argument
public static int pow(int base, int exp, int productSoFar)
{
   if (exp == 0)
      return productSoFar;
   else
      return pow(base, exp - 1, base * productSoFar);
}
```

Note that the only differences between implementation #2 and implementation #1 are:

- We added a third parameter to #2
- We return that third parameter's value instead of `1` in the base case
- We have a third argument – a multiplication result – passed as the argument to that third parameter when we make our recursive call

Below we have the "method notecards" generated by both our implementations, *before* the point where the base case returns. Note that the first two arguments are the same in both cases, for each step – but the second implementation has a third parameter whose value steadily increases with each method call, until by the time we reach the base case, that parameter's value is our answer. We calculate the third argument for each call by multiplying the `base` by the `productSoFar`.

```
Implementation #1                        Implementation #2
--------------------                    --------------------------

-----------------                       ------------------------
| main                                  | main
|   int x;                              |    int x;
|   x = pow(3, 4);                      |    x = pow(3, 4, 1);
|                                       |
|                                       |
|________________                       |_________________________
| pow     (base: 3)                     | pow         (base: 3)
|  (#1)   (exp: 4)                      |  (#1)       (exp: 4)
|                                       |             (productSoFar: 1)
|  need: pow(3, 3)                      | need: pow(3, 3, 3)
|_________________                      |_________________________
| pow      (base: 3)                    | pow         (base: 3)
|  (#2)    (exp: 3)                     |  (#2)       (exp: 3)
|                                       |             (productSoFar: 3)
| need: pow(3, 2)                       | need: pow(3, 2, 9)
|____________                           |__________________________
| pow      (base: 3)                    | pow         (base: 3)
| (#3)     (exp: 2)                     |  (#3)       (exp: 2)
|                                       |             (productSoFar: 9)
|  need: pow(3, 1)                      | need: pow(3, 1, 27)
|____________                           |__________________________
| pow      (base: 3)                    | pow         (base: 3)
|  (#4)    (exp: 1)                     |  (#4)       (exp: 1)
|                                       |             (productSoFar: 27)
|  need: pow(3, 0)                      | need: pow(3, 0, 81)
|____________                           |__________________________
| pow       (base: 3)                   | pow         (base: 3)
|   (#5)    (exp: 0)                    |  (#5)       (exp: 0)
|                                       |             (productSoFar: 81)
|  base case!                           |  base case!
|____________                           |___________________________
```

Next, we can indicate what gets returned as the method calls above return one by one (the base case, of course, is the first method to return, then call #4, then call #3, then call #2, and finally call #1). Note that in Implementation #1, we have to do the multiplications as we complete each recursive call, since the `else` case for Implementation #1 was:

```
return base * pow(base, exp-1);
```

but in Implementation #2, we already did the needed multiplication *before* making the recursive call, since the `else` case for Implementation #2 was:

```
return pow(base, exp - 1, base * productSoFar);
```

and so for Implementation #2 we are not doing any calculations as we return, but merely returning the same value, `81`, that we've already calculated as our solution:

```
Implementation #1
--------------------

-----------------
| main
|   int x;
|   x = pow(3, 4);
|     // once pow(3, 4) returns,
|     // 81 is stored in x
|________________
| pow     (base: 3)
|  (#1)   (exp: 4)
|           ------> return 81
|  need: pow(3, 3)
|_________________
| pow      (base: 3)
|  (#2)    (exp: 3)
|           ------> return 27
| need: pow(3, 2)
|____________
| pow      (base: 3)
| (#3)     (exp: 2)
|           ------> return 9
|  need: pow(3, 1)
|____________
| pow      (base: 3)
|  (#4)    (exp: 1)
|           ------> return 3
|  need: pow(3, 0)
|____________
| pow       (base: 3)
|   (#5)    (exp: 0)
|            ------> return 1
|  base case!
|____________
```

```
Implementation #2
--------------------------

------------------------
| main
|    int x;
|    x = pow(3, 4, 1);
|     // once pow(3, 4, 1) returns,
|     // 81 is stored in x
|_________________________
| pow          (base: 3)
|  (#1)        (exp: 4)
|              (productSoFar: 1)
| need: pow(3, 3, 3)    ----> return 81
|_________________________
| pow          (base: 3)
|  (#2)        (exp: 3)
|              (productSoFar: 3)
| need: pow(3, 2, 9)    ----> return 81
|__________________________
| pow          (base: 3)
|  (#3)        (exp: 2)
|              (productSoFar: 9)
| need: pow(3, 1, 27)   ----> return 81
|__________________________
| pow          (base: 3)
|  (#4)        (exp: 1)
|              (productSoFar: 27)
| need: pow(3, 0, 81)   ----> return 81
|__________________________
| pow          (base: 3)
|  (#5)        (exp: 0)
|              (productSoFar: 81)
|  base case!           ----> return 81
|___________________________
```

To convert this to a loop implementation, let's first put the accumulator-recursive code into the layout we mentioned in the last lecture packet:

```
  recursiveMethod(params)
  {
     if (recursiveCase)
     {
        code we run before recursive call
        recursiveMethod(args);
     }
     else // base case
     {
        base case code
     }
  }
```

That is, convert this:

```
public static int pow(int base, int exp, int productSoFar)
{
   if (exp == 0)
      return productSoFar;
   else
      return pow(base, exp - 1, base * productSoFar);
}
```

to this:

```
public static int pow(int base, int exp, int productSoFar)
{
   if (exp > 0)
      return pow(base, exp - 1, base * productSoFar);
   else  // exp == 0
      return productSoFar;
}
```

by switching the `if` and `else` cases. Now, we can use our general tail-recursion-to-loop-implementation technique to convert this into a loop implementation:

```
public static int pow(int base, int exp, int productSoFar)
{
   while (exp > 0)
   {
      // There was code before recursive call.
      // The statement
      //      return pow(base, exp - 1, base * productSoFar);
      // is converted into the assignments below:
      base = base;
      exp = exp - 1;
      productSoFar = base * productSoFar;
   }
   return productSoFar;
}
```

We don't need to assign `base` to itself, so we can remove that assignment:

```
public static int pow(int base, int exp, int productSoFar)
{
   while (exp > 0)
   {
      exp = exp - 1;
      productSoFar = base * productSoFar;
   }
   return productSoFar;
}
```

And with that, we have a working loop implementation! Of course, we are still demanding that the clients pass in a `1` as the third argument; we could make `productSoFar` a local variable, if we wanted, in order to avoid making the clients pass in an initial value for that variable:

```
public static int pow(int base, int exp)
{
   int productSoFar = 1;
   while (exp > 0)
   {
      exp = exp - 1;
      productSoFar = base * productSoFar;
   }
   return productSoFar;
}
```

Now, there is an important thing to take from this example – all three of our implementations are implementing the *exact same algorithm.* In all three cases, we calculate an exponentiation by multiplying the *base*, by an existing exponentiation with the same `base` and one smaller `exp`. And as a result, in terms of everything but method calls, they all do the exact same work.

Consider the calculation of 3^4, for example. Whether you look at the mathematical formula, or any of the three code examples (the forward-recursive version, the accumulator-recursive version, or the loop version), we always need to check the exponent to see if it's zero or not, and if not, then the recursive subproblem will have an exponent one lower in value. And so we will compare the exponent to zero a total of five times – when the exponent is 4, when it is 3, when it is 2, when it is 1, and when it is 0. Whether you look at the pure math, or any of the code examples, you will need five comparisons as you go through the calculation.

Furthermore, for four of those cases – all the recursive cases – you will need to do a subtraction (to get an exponent one lower than your current exponent) and a multiplication (to multiply the base by some existing product). The base case doesn't perform the multiplication or the subtraction, but the recursive case does, so if you run the recursive case four times (when the exponent is 4, when it is 3, when it is 2, and when it is 1), you will need to perform four subtractions and four multiplications.

Go ahead and trace through the calculation of 3^4 using the pure math, and then each of the three coded implementations. You will see that each time, we are performing five comparisons, four multiplications, and four subtractions. It's the same algorithm each time, so the same work is being done each time. The same was true of the factorial algorithm earlier in this notes packet.

The one big difference is that the recursive versions use the extra assistance of method calls to keep track of the values of the base and exponent, and the non-forward-recursive versions use an extra variable to store a temporary product. But the basic computation is the same as we move from recursive code to loop-based code. The fundamental calculations needed – the comparisons, subtractions, and multiplications – don't change no matter how we code the algorithm. If we had a *different* algorithm, though, that could change – because then we'd be changing the fundamental computational process we were using, and not just changing the way in which we coded it. We will explore that idea, a bit later in the semester.