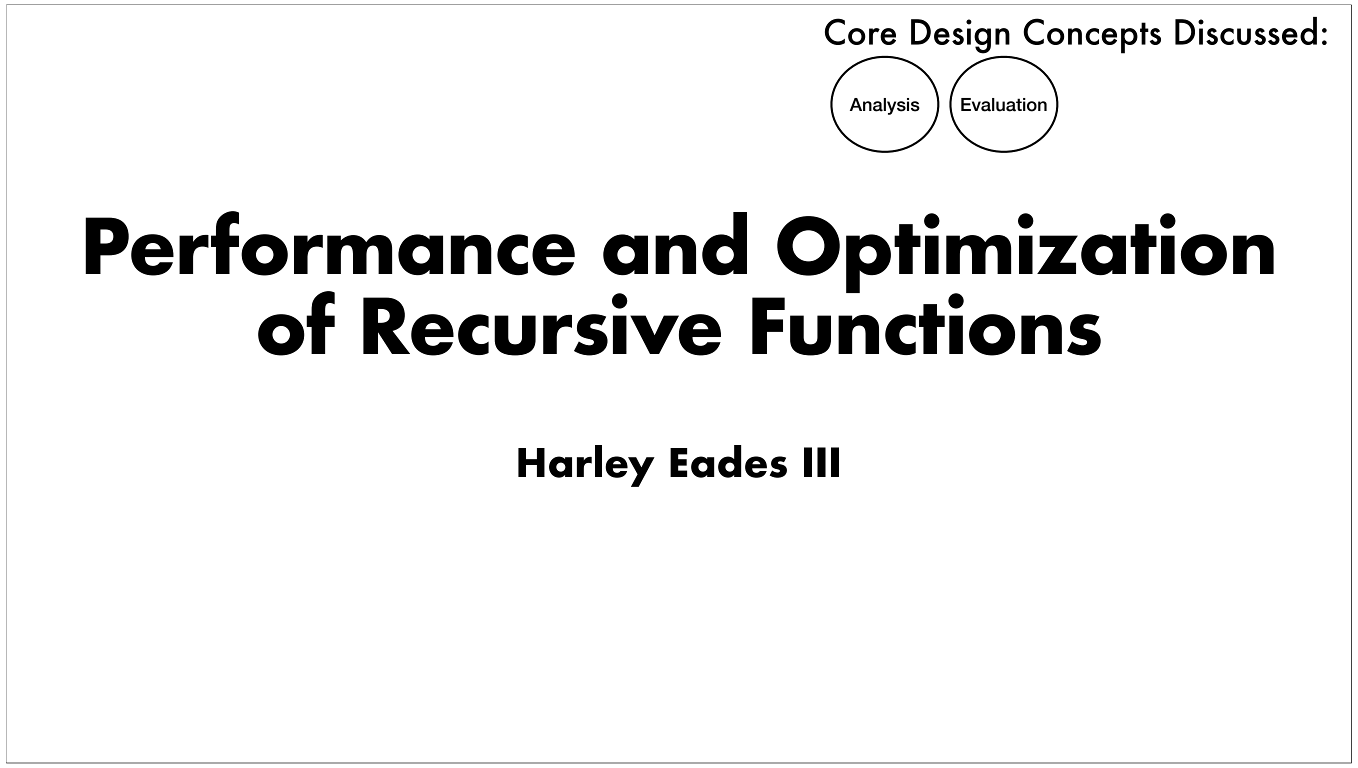


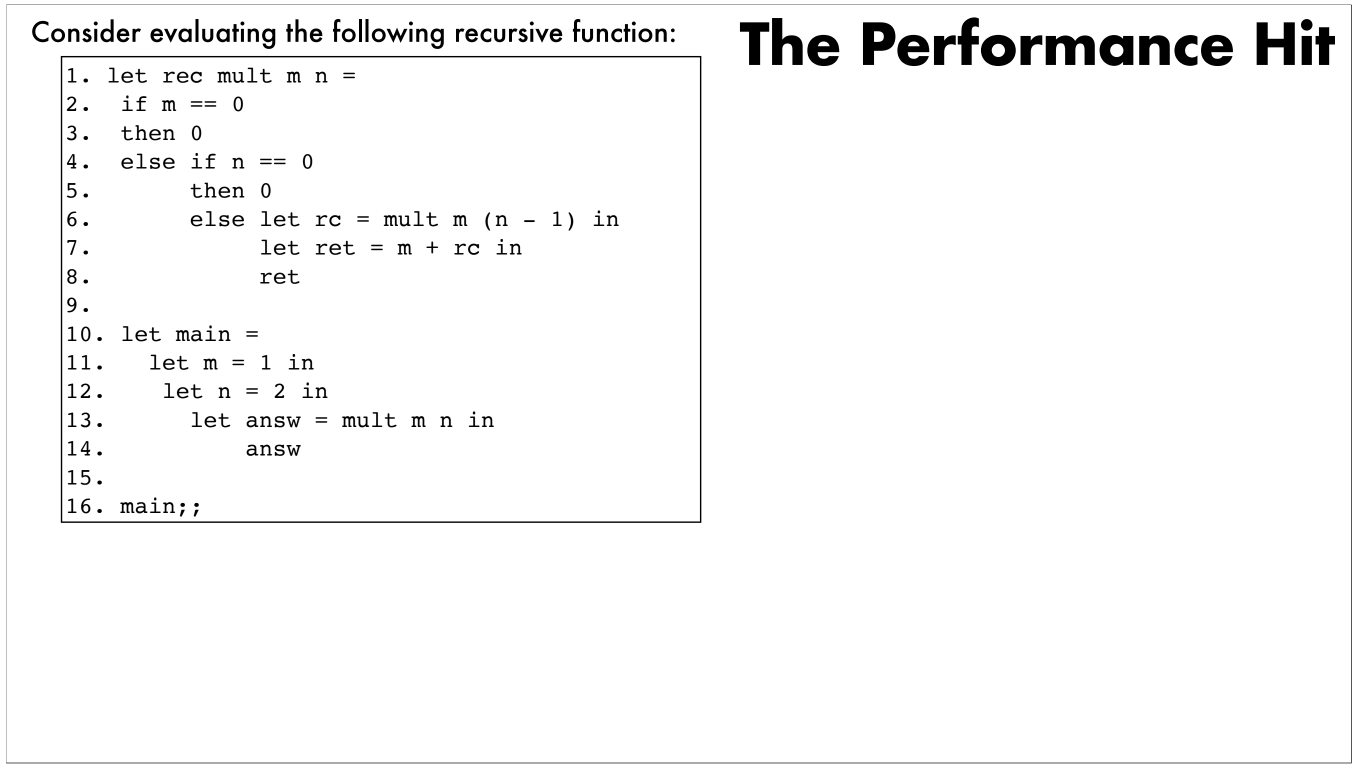
Core Design Concepts Discussed:

Analysis Evaluation

Performance and Optimization of Recursive Functions

Harley Eades III

In this lecture we will be looking at how the structure of a recursive function can greatly affect its performance. This performance hit is caused by the way compilers implement recursion using activation records. We will then look at how we can prevent this performance problem using tail-call optimization. First, let's try and understand the problem...

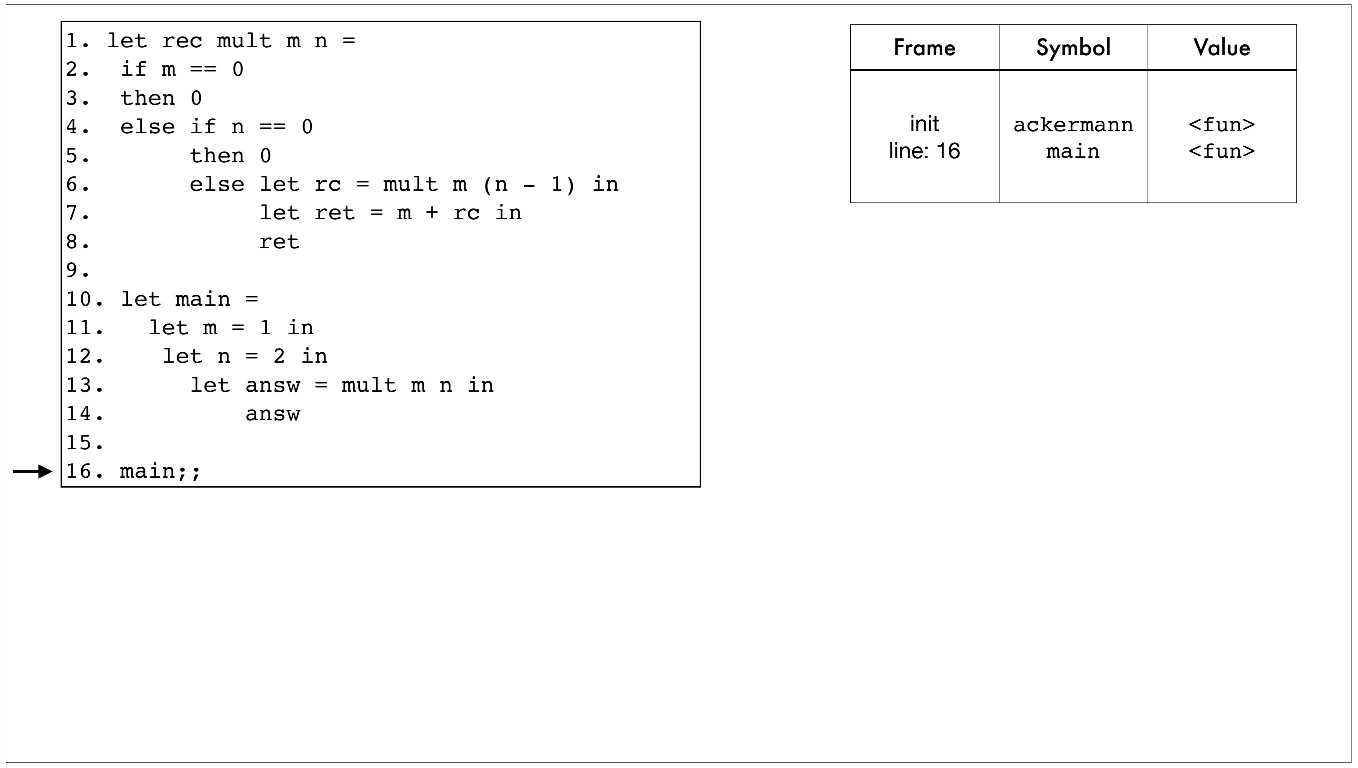


The Performance Hit

Consider evaluating the following recursive function:

```
1. let rec mult m n =
2.  if m == 0
3.  then 0
4.  else if n == 0
5.       then 0
6.       else let rc = mult m (n - 1) in
7.            let ret = m + rc in
8.            ret
9.
10. let main =
11.   let m = 1 in
12.    let n = 2 in
13.      let answ = mult m n in
14.          answ
15.
16. main;;
```

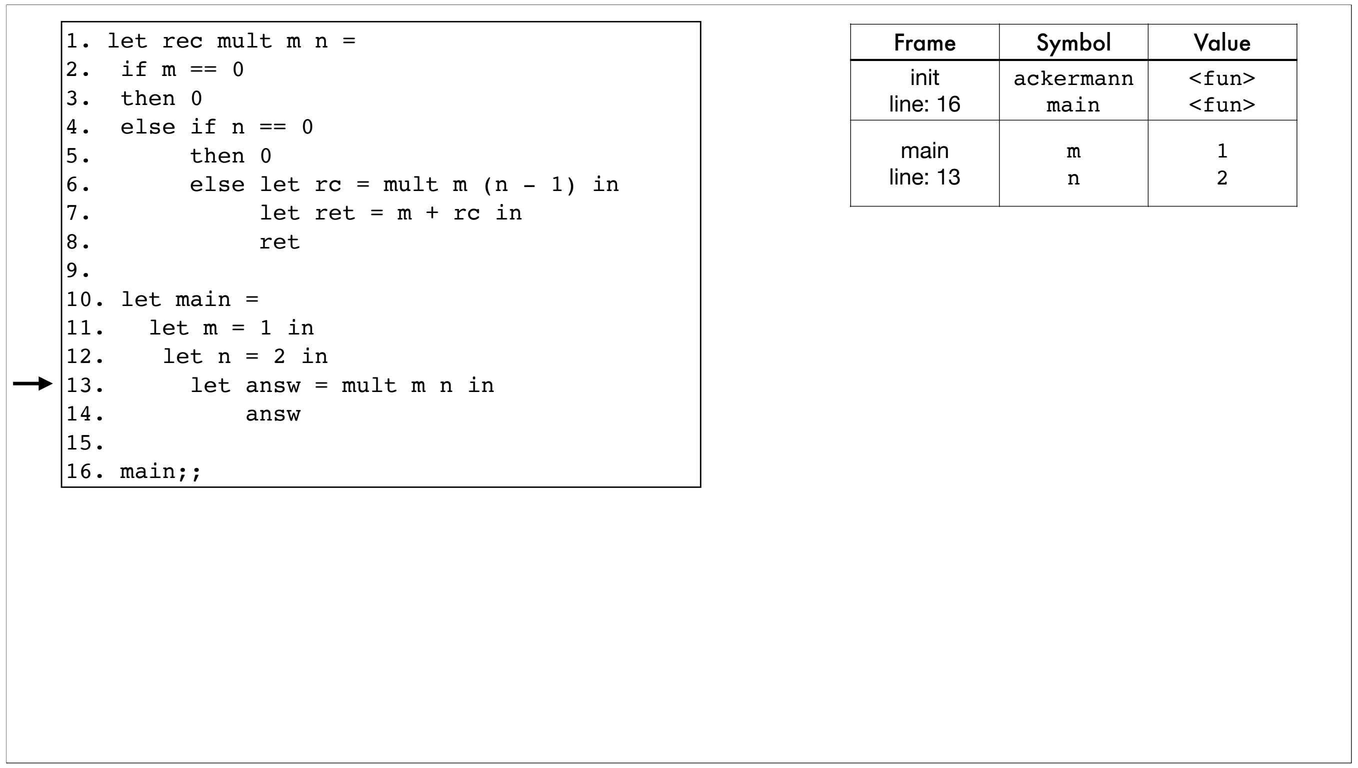
Consider this implementation of multiplication using addition. We are to evaluate line 16 which is a call to main...



```
   1. let rec mult m n =
   2.  if m == 0
   3.  then 0
   4.  else if n == 0
   5.       then 0
   6.       else let rc = mult m (n - 1) in
   7.            let ret = m + rc in
   8.            ret
   9.
   10. let main =
   11.   let m = 1 in
   12.    let n = 2 in
   13.      let answ = mult m n in
   14.          answ
   15.
→  16. main;;
```

Frame	Symbol	Value
init line: 16	ackermann main	<fun> <fun>

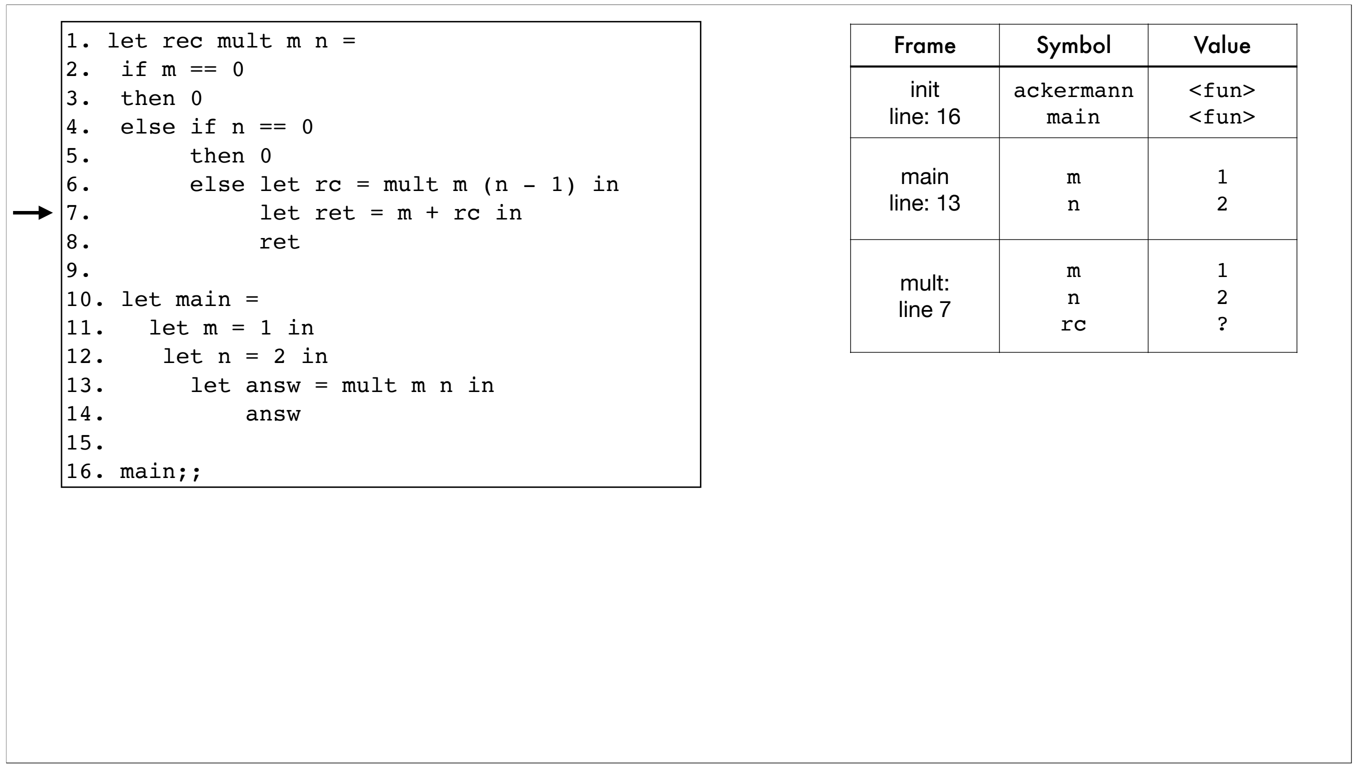
Now to evaluate main we must evaluate the function call to multi on line 13. So we push an activation record onto the call stack...



```
1. let rec mult m n =
2.  if m == 0
3.  then 0
4.  else if n == 0
5.       then 0
6.       else let rc = mult m (n - 1) in
7.            let ret = m + rc in
8.            ret
9.
10. let main =
11.   let m = 1 in
12.    let n = 2 in
13.      let answ = mult m n in
14.          answ
15.
16. main;;
```

Frame	Symbol	Value
init line: 16	ackermann main	<fun> <fun>
main line: 13	m n	1 2

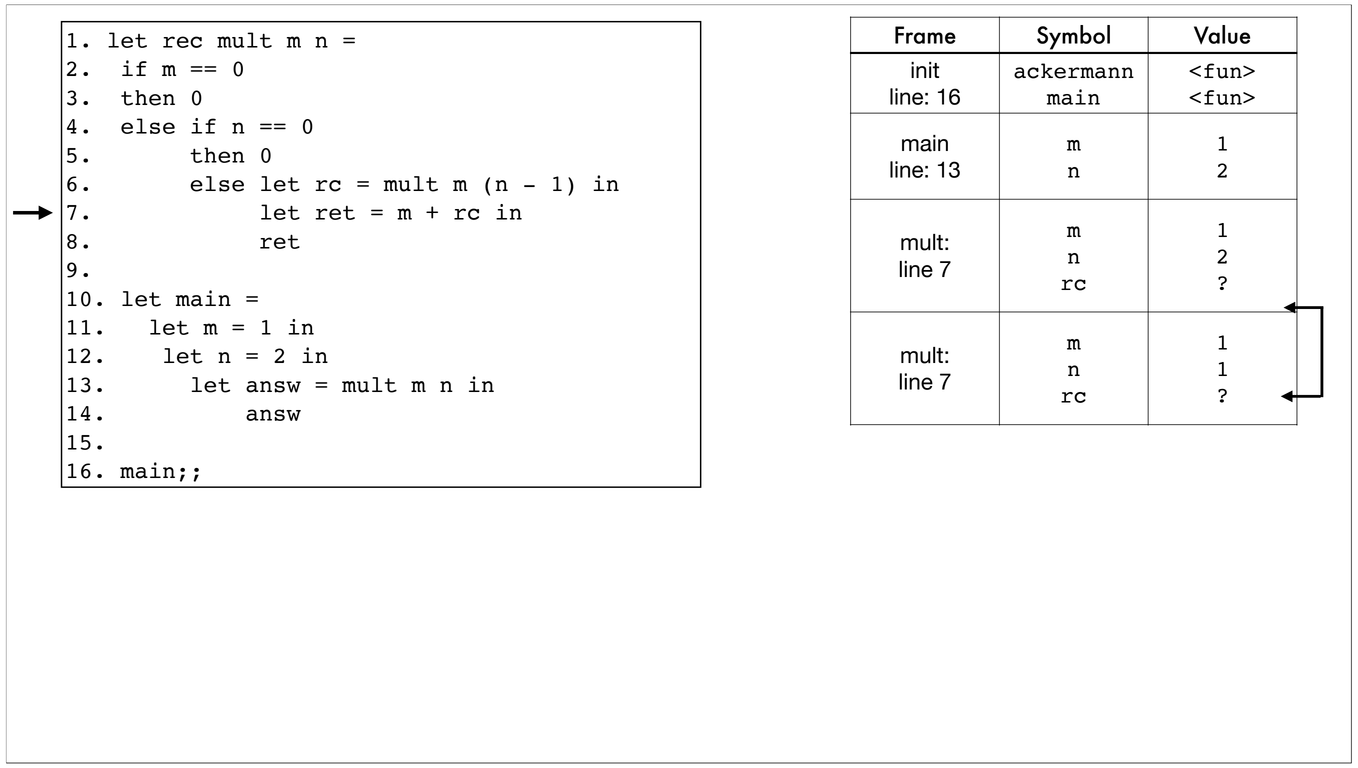
In this activation record we have bindings for m, n, we are not going to keep track of return values in the activation record, because this is not usually done. In order to obtain a value for answ we must first evaluate mult, which based on the values for m and n we will add an activation record for the function call on line 7...



```
1. let rec mult m n =
2.  if m == 0
3.  then 0
4.  else if n == 0
5.       then 0
6.       else let rc = mult m (n - 1) in
7.            let ret = m + rc in
8.            ret
9.
10. let main =
11.   let m = 1 in
12.    let n = 2 in
13.      let answ = mult m n in
14.          answ
15.
16. main;;
```

Frame	Symbol	Value
init line: 16	ackermann main	<fun> <fun>
main line: 13	m n	1 2
mult: line 7	m n rc	1 2 ?

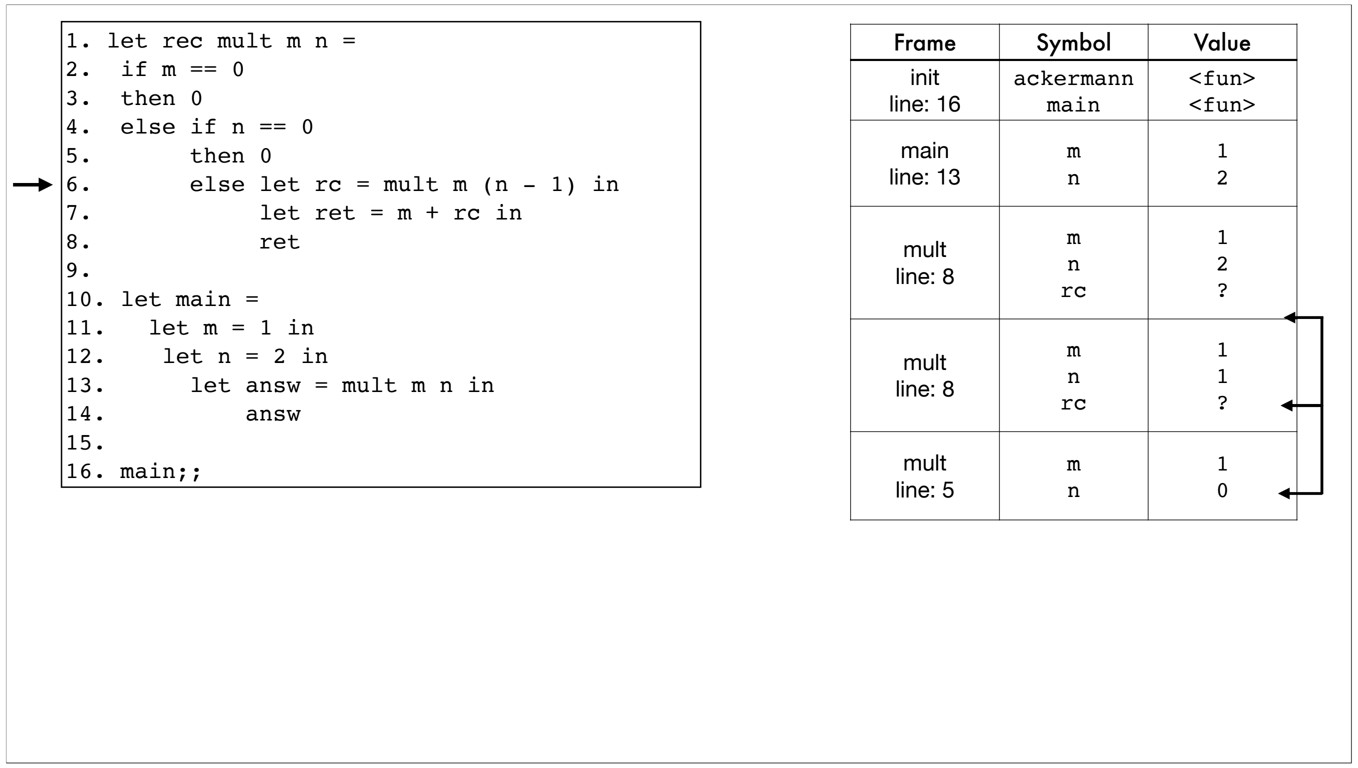
In order to find a value for answ, we have to find a value for ret, but this depends on rc which is a recursive call to mult. It is this recursive call that is going to result in a performance problem. First, we add a new activation record for the call to mult...



```
1. let rec mult m n =
2.  if m == 0
3.  then 0
4.  else if n == 0
5.       then 0
6.       else let rc = mult m (n - 1) in
7.            let ret = m + rc in
8.            ret
9.
10. let main =
11.   let m = 1 in
12.    let n = 2 in
13.      let answ = mult m n in
14.          answ
15.
16. main;;
```

Frame	Symbol	Value
init line: 16	ackermann main	<fun> <fun>
main line: 13	m n	1 2
mult: line 7	m n rc	1 2 ?
mult: line 7	m n rc	1 1 ?

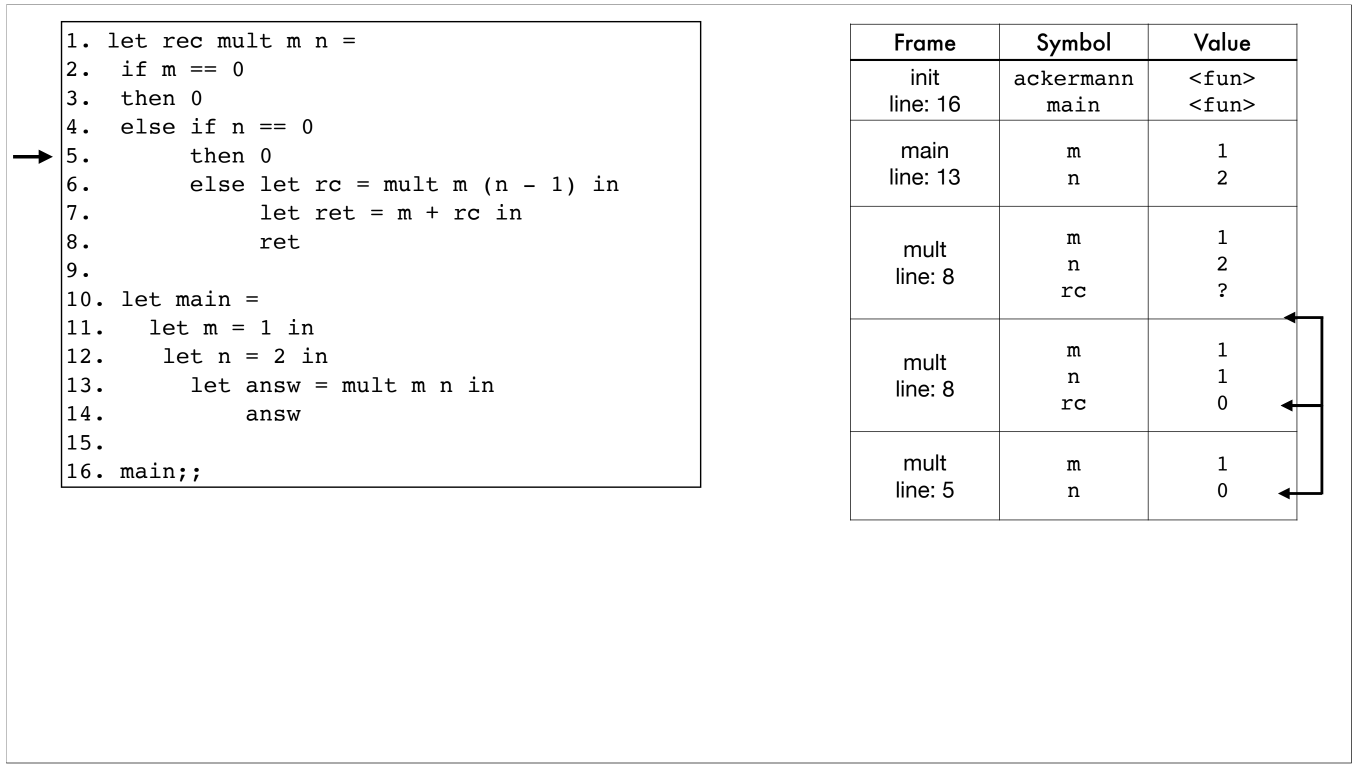
Take notice that the bindings of the record above our current record is dependent upon the return value of the lower record. That is, we cannot compute the value for rc in the record above without computing the value of ret in the record below. This dependency is the cause of a performance issue, because we cannot discard any of these records until we have computed later records...



```
1. let rec mult m n =
2.  if m == 0
3.  then 0
4.  else if n == 0
5.       then 0
6.       else let rc = mult m (n - 1) in
7.            let ret = m + rc in
8.            ret
9.
10. let main =
11.   let m = 1 in
12.    let n = 2 in
13.      let answ = mult m n in
14.          answ
15.
16. main;;
```

Frame	Symbol	Value
init line: 16	ackermann main	<fun> <fun>
main line: 13	m n	1 2
mult line: 8	m n rc	1 2 ?
mult line: 8	m n rc	1 1 ?
mult line: 5	m n	1 0

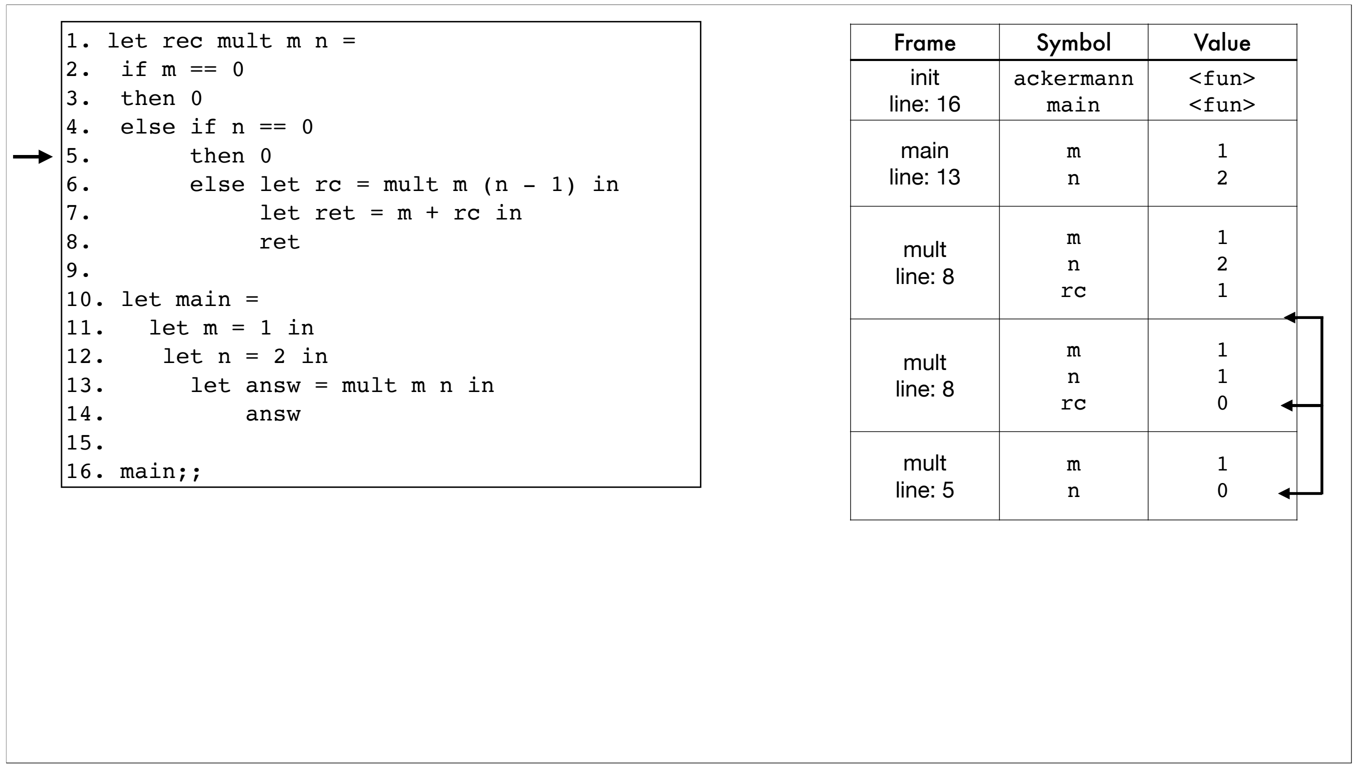
Next we must add another activation record for our call to mult on line 6, but this time we hit a base case. Thus, this activation record is popped, and it's return value is added to the record above updating rc...



```
1. let rec mult m n =
2.  if m == 0
3.  then 0
4.  else if n == 0
5.       then 0
6.       else let rc = mult m (n - 1) in
7.            let ret = m + rc in
8.            ret
9.
10. let main =
11.   let m = 1 in
12.    let n = 2 in
13.      let answ = mult m n in
14.          answ
15.
16. main;;
```

Frame	Symbol	Value
init line: 16	ackermann main	<fun> <fun>
main line: 13	m n	1 2
mult line: 8	m n rc	1 2 ?
mult line: 8	m n rc	1 1 0
mult line: 5	m n	1 0

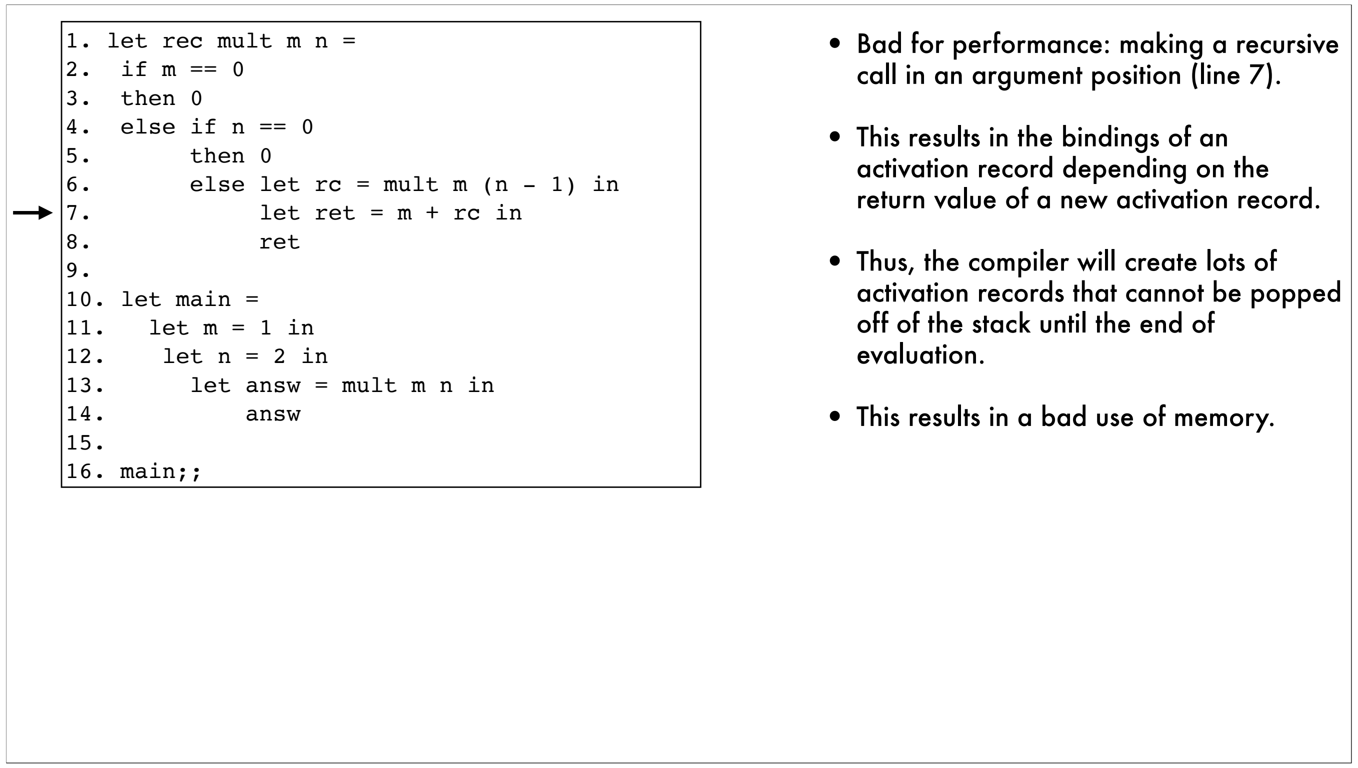
We then keep popping and updating the records above. We have one more update to do...



```
1. let rec mult m n =
2.  if m == 0
3.  then 0
4.  else if n == 0
5.       then 0
6.       else let rc = mult m (n - 1) in
7.            let ret = m + rc in
8.            ret
9.
10. let main =
11.   let m = 1 in
12.    let n = 2 in
13.      let answ = mult m n in
14.          answ
15.
16. main;;
```

Frame	Symbol	Value
init line: 16	ackermann main	<fun> <fun>
main line: 13	m n	1 2
mult line: 8	m n rc	1 2 1
mult line: 8	m n rc	1 1 0
mult line: 5	m n	1 0

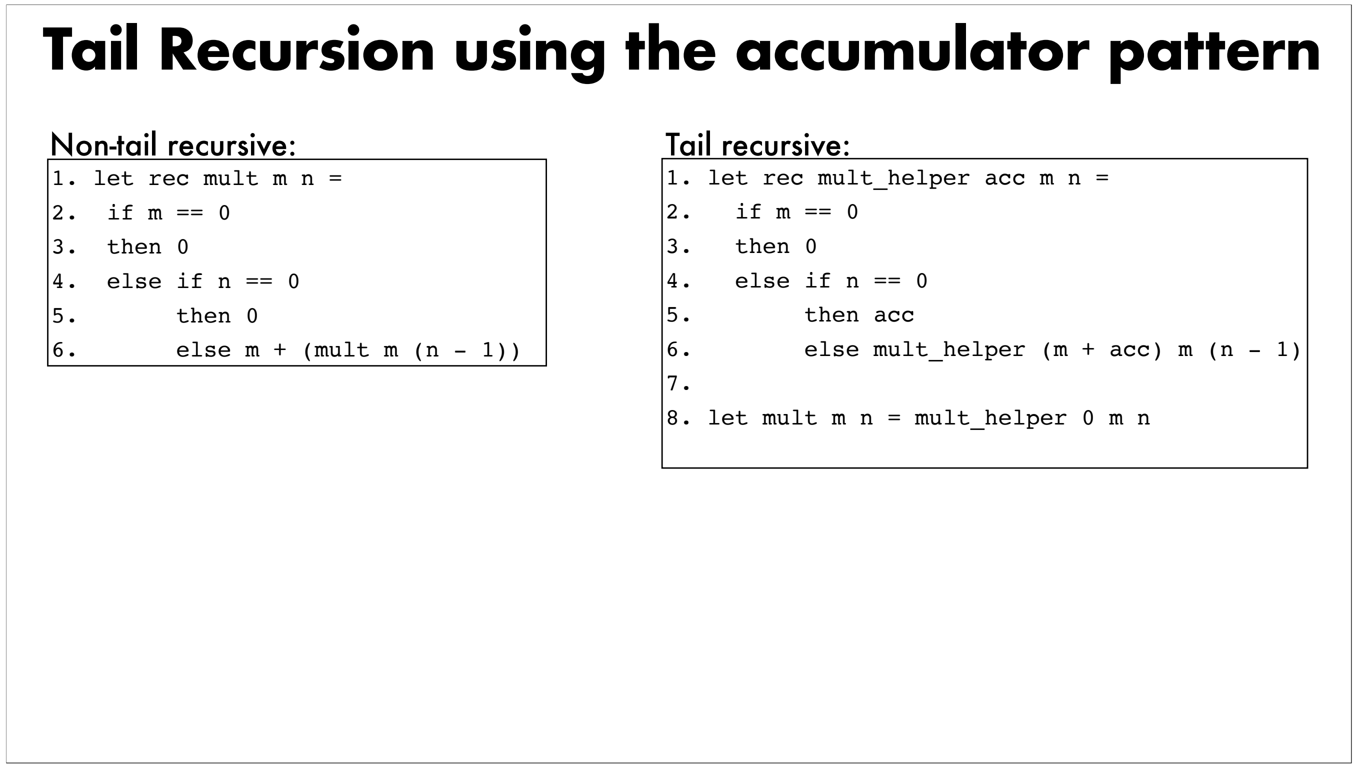
At this point the stack as been unwound, and we can return our value of 2. Now this dependency we have uncovered results in activation records being on the stack that cannot be removed until evaluation is complete. For large input values, the stack will grow and grow, and often, result in a stack overflow...



```
   1. let rec mult m n =
   2.  if m == 0
   3.  then 0
   4.  else if n == 0
   5.       then 0
   6.       else let rc = mult m (n - 1) in
→  7.            let ret = m + rc in
   8.            ret
   9.
   10. let main =
   11.   let m = 1 in
   12.    let n = 2 in
   13.      let answ = mult m n in
   14.          answ
   15.
   16. main;;
```

- Bad for performance: making a recursive call in an argument position (line 7).
- This results in the bindings of an activation record depending on the return value of a new activation record.
- Thus, the compiler will create lots of activation records that cannot be popped off of the stack until the end of evaluation.
- This results in a bad use of memory.

So let's sum up. The performance problem is a result of poorly using the stack during evaluation. If an argument to a function is a recursive call, then the bindings of the current activation record cannot be computed without creating more activation records, and the activation records cannot be popped off of the stack until the end of evaluation. This can cause memory overflow and is a source of a lot of bugs! So how do we fix this problem?



Tail Recursion using the accumulator pattern

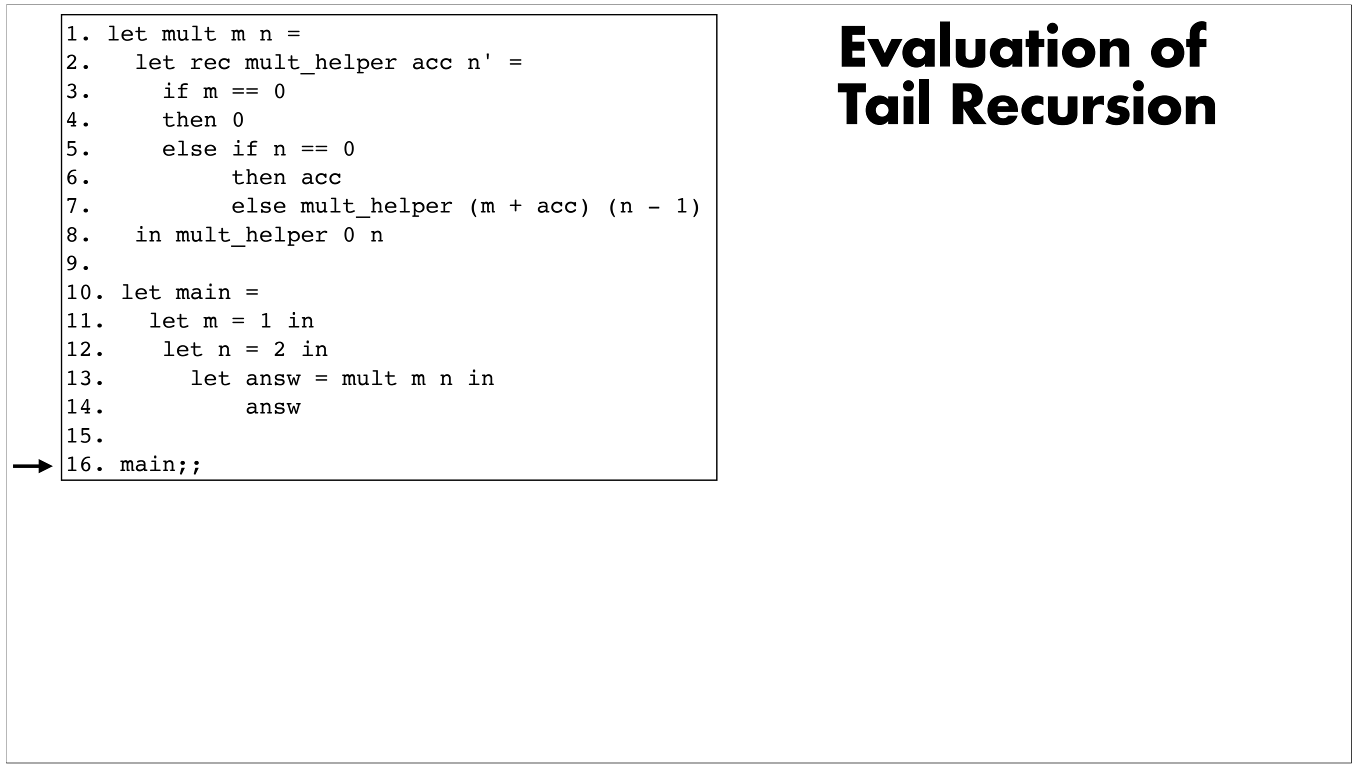
Non-tail recursive:

```
1. let rec mult m n =
2.  if m == 0
3.  then 0
4.  else if n == 0
5.       then 0
6.       else m + (mult m (n - 1))
```

Tail recursive:

```
1. let rec mult_helper acc m n =
2.   if m == 0
3.   then 0
4.   else if n == 0
5.        then acc
6.        else mult_helper (m + acc) m (n - 1)
7.
8. let mult m n = mult_helper 0 m n
```

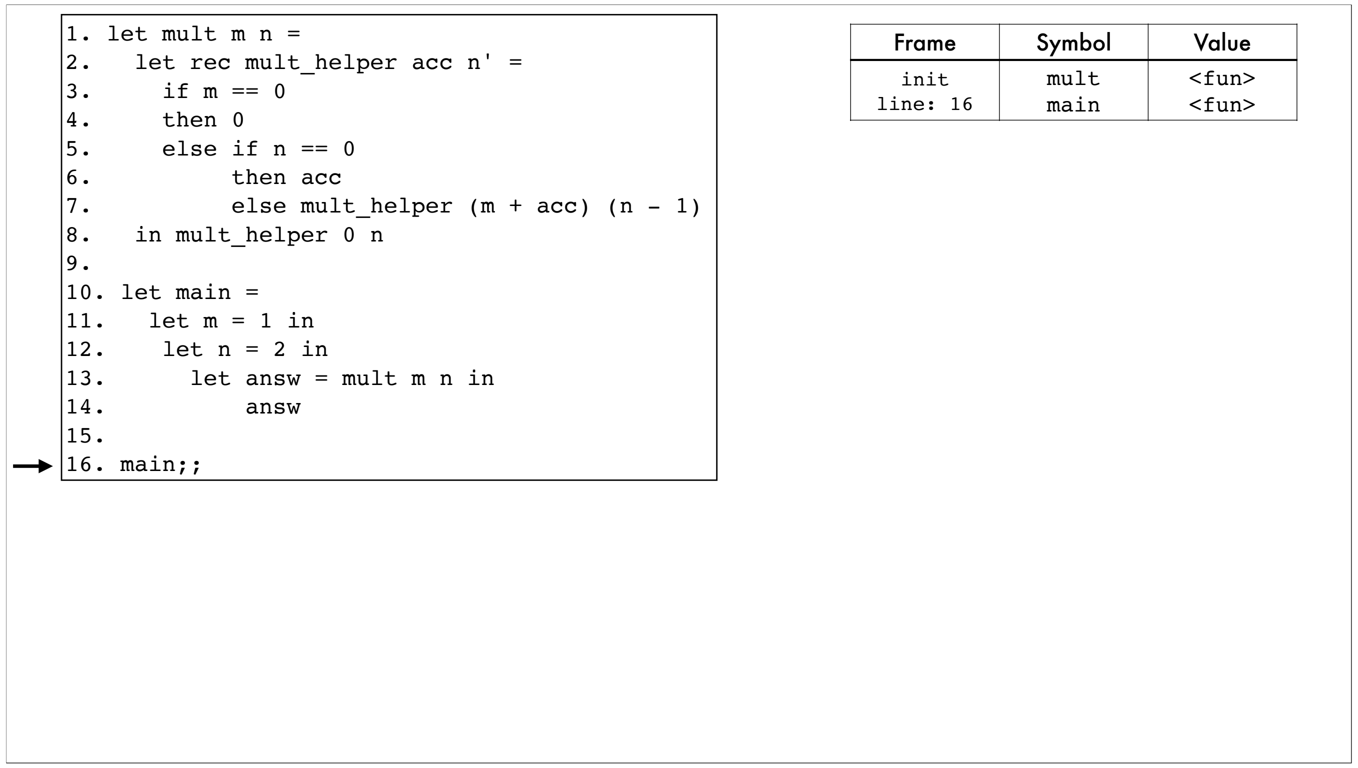
The left side is the definition of mult we have been studying, but slightly simplified by removing the redundant lets. Then I converted this function into a tail recursive function using a helper function called mult_helper on the right. This helper function takes in three arguments: what is called the accumulator, here named acc, and the values we are multiplying together called m and n. The accumulator can be thought of as a new global variable that is keeping track of our return value, that is the repeated addition of m. Since we are doing recursion over n, in the base case, line 5, when n is 0, we simply return the value of the accumulator. This is why on line 8, we initialize the accumulator to 0, so that in the case when we get 0 for n right away, we will return 0. Finally, in the step case, line 6, we add m to the accumulator and make a "tail recursive call" to mult_helper. Compare this line to line 6 of the non-tail recursive function on the left. We esstenally have pushed the addition into the argument position pulling the recursive call out. But, we are computing the same value. The tail recursive version on the right can be slightly simplified into....



Evaluation of Tail Recursion

```
1. let mult m n =
2.   let rec mult_helper acc n' =
3.     if m == 0
4.     then 0
5.     else if n == 0
6.          then acc
7.          else mult_helper (m + acc) (n - 1)
8.   in mult_helper 0 n
9.
10. let main =
11.   let m = 1 in
12.    let n = 2 in
13.      let answ = mult m n in
14.          answ
15.
16. main;;
```

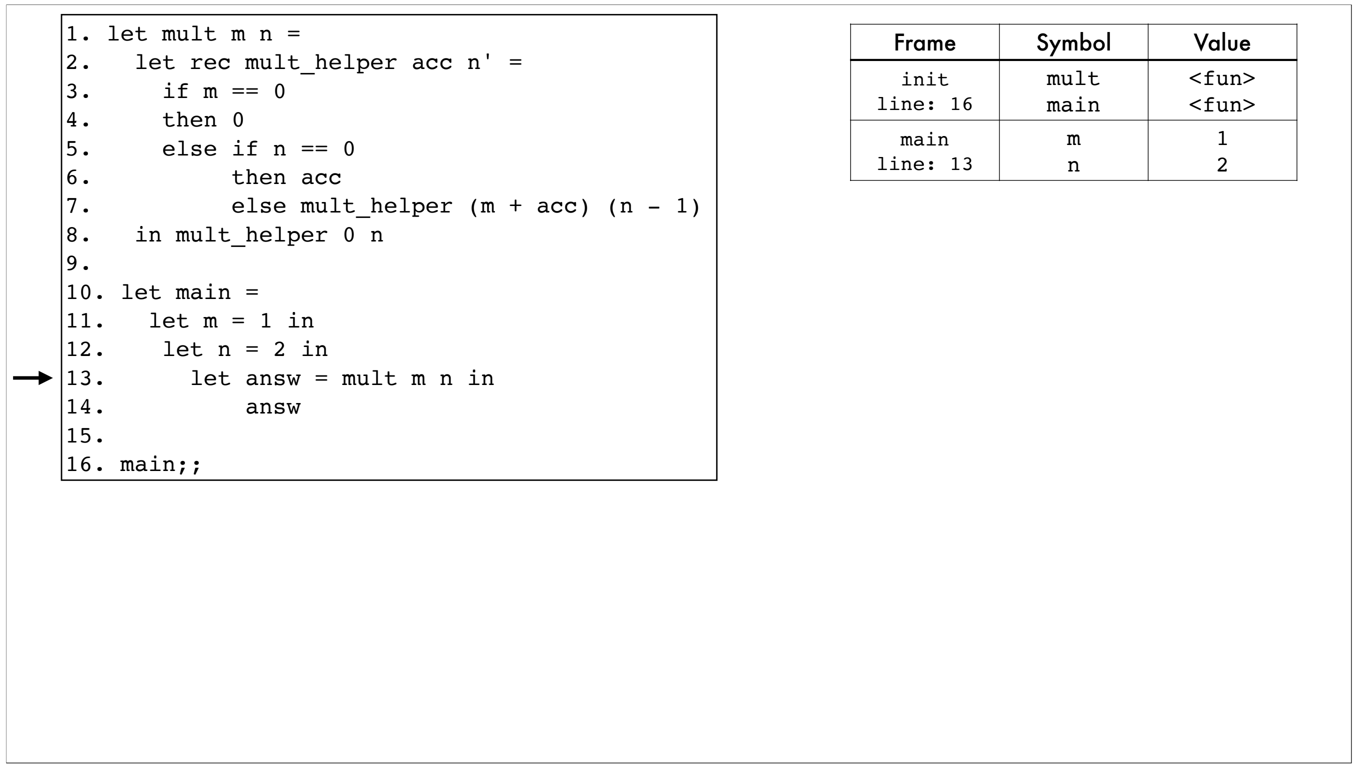
The version above. We simply nested the let syntax resulting in the helper function becoming local to mult. This is very common in OCaml. Now lets evaluate this version using activation records and see if we encounter the same sort of dependency we saw earlier. First, we must evaluate main on line 16. This will in turn require us to evaluate line 13 of main, the first function call to mult...



```
   1. let mult m n =
   2.   let rec mult_helper acc n' =
   3.     if m == 0
   4.     then 0
   5.     else if n == 0
   6.          then acc
   7.          else mult_helper (m + acc) (n - 1)
   8.   in mult_helper 0 n
   9.
   10. let main =
   11.   let m = 1 in
   12.    let n = 2 in
   13.      let answ = mult m n in
   14.          answ
   15.
→ 16. main;;
```

Frame	Symbol	Value
init line: 16	mult main	<fun> <fun>

The version above. We simply nested the let syntax resulting in the helper function becoming local to mult. This is very common in OCaml. Now lets evaluate this version using activation records and see if we encounter the same sort of dependency we saw earlier. First, we must evaluate main on line 16. This will in turn require us to evaluate line 13 of main, the first function call to mult...



```
  1. let mult m n =
  2.   let rec mult_helper acc n' =
  3.     if m == 0
  4.     then 0
  5.     else if n == 0
  6.          then acc
  7.          else mult_helper (m + acc) (n - 1)
  8.   in mult_helper 0 n
  9.
  10. let main =
  11.   let m = 1 in
  12.    let n = 2 in
→ 13.      let answ = mult m n in
  14.          answ
  15.
  16. main;;
```

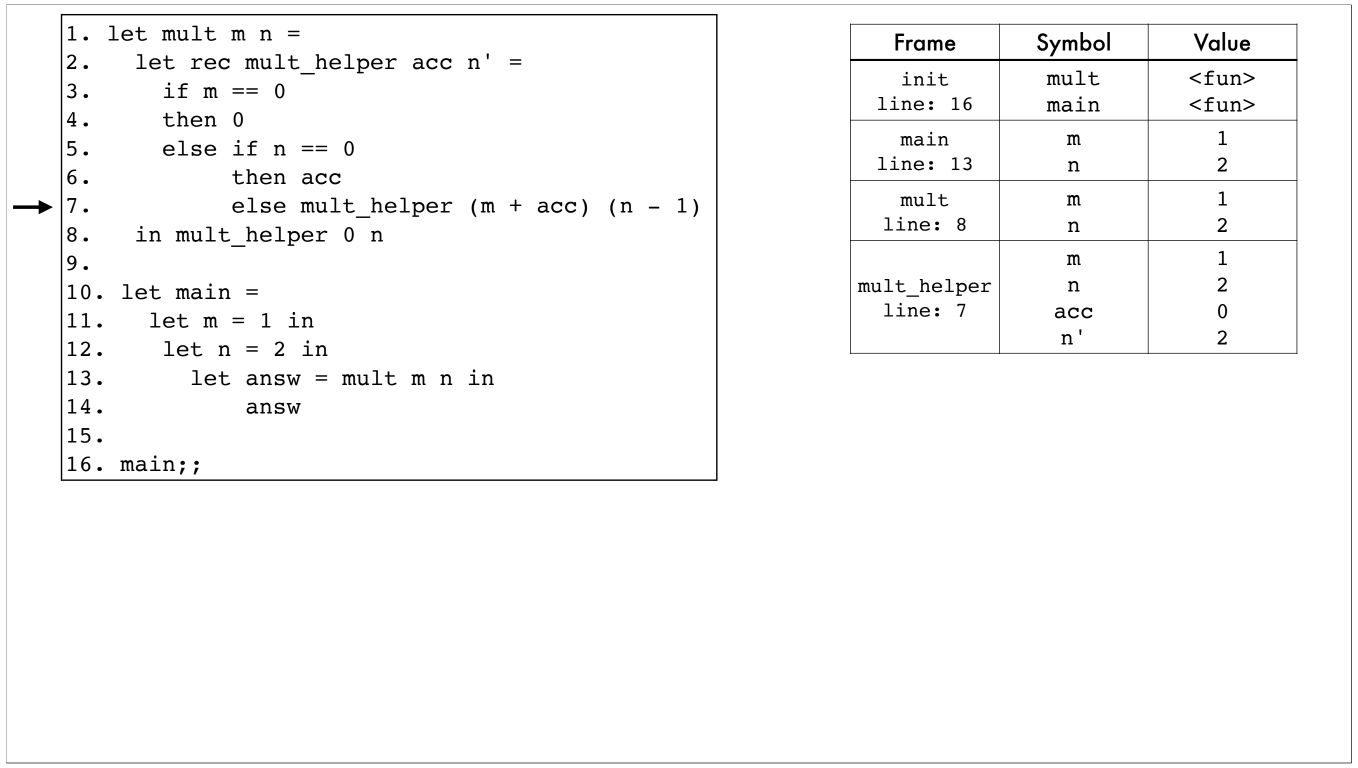
Frame	Symbol	Value
init line: 16	mult main	<fun> <fun>
main line: 13	m n	1 2

Given the value of m and n, we will then be required to evaluate the call to mult_helper on line 8...

```
   1. let mult m n =
   2.   let rec mult_helper acc n' =
   3.     if m == 0
   4.     then 0
   5.     else if n == 0
   6.          then acc
   7.          else mult_helper (m + acc) (n - 1)
→  8.   in mult_helper 0 n
   9.
   10. let main =
   11.   let m = 1 in
   12.    let n = 2 in
   13.      let answ = mult m n in
   14.          answ
   15.
   16. main;;
```

Frame	Symbol	Value
init line: 16	mult main	<fun> <fun>
main line: 13	m n	1 2
mult line: 8	m n	1 2

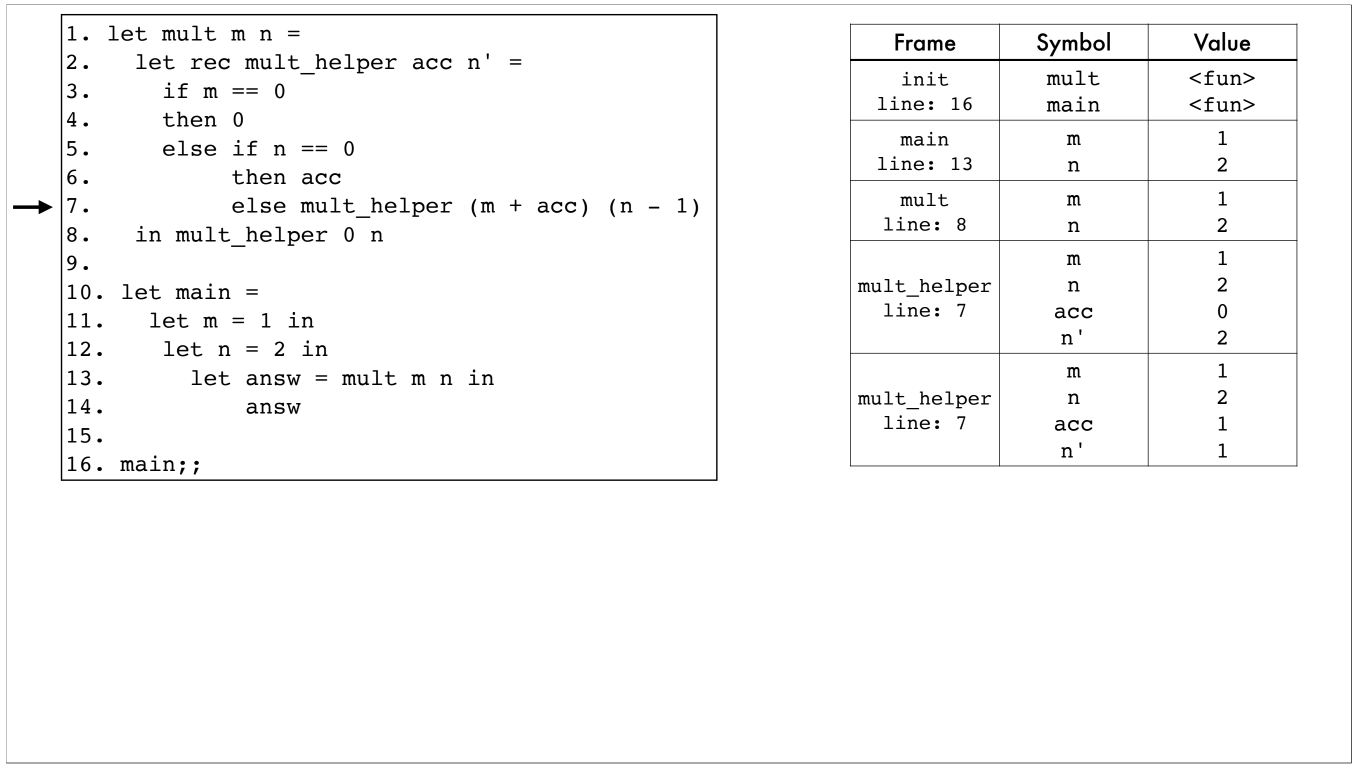
Next we add a new activation record to evaluate mult_helper on line 8...



```
  1. let mult m n =
  2.   let rec mult_helper acc n' =
  3.     if m == 0
  4.     then 0
  5.     else if n == 0
  6.          then acc
→ 7.          else mult_helper (m + acc) (n - 1)
  8.   in mult_helper 0 n
  9.
  10. let main =
  11.   let m = 1 in
  12.    let n = 2 in
  13.      let answ = mult m n in
  14.          answ
  15.
  16. main;;
```

Frame	Symbol	Value
init line: 16	mult main	<fun> <fun>
main line: 13	m n	1 2
mult line: 8	m n	1 2
mult_helper line: 7	m n acc n'	1 2 0 2

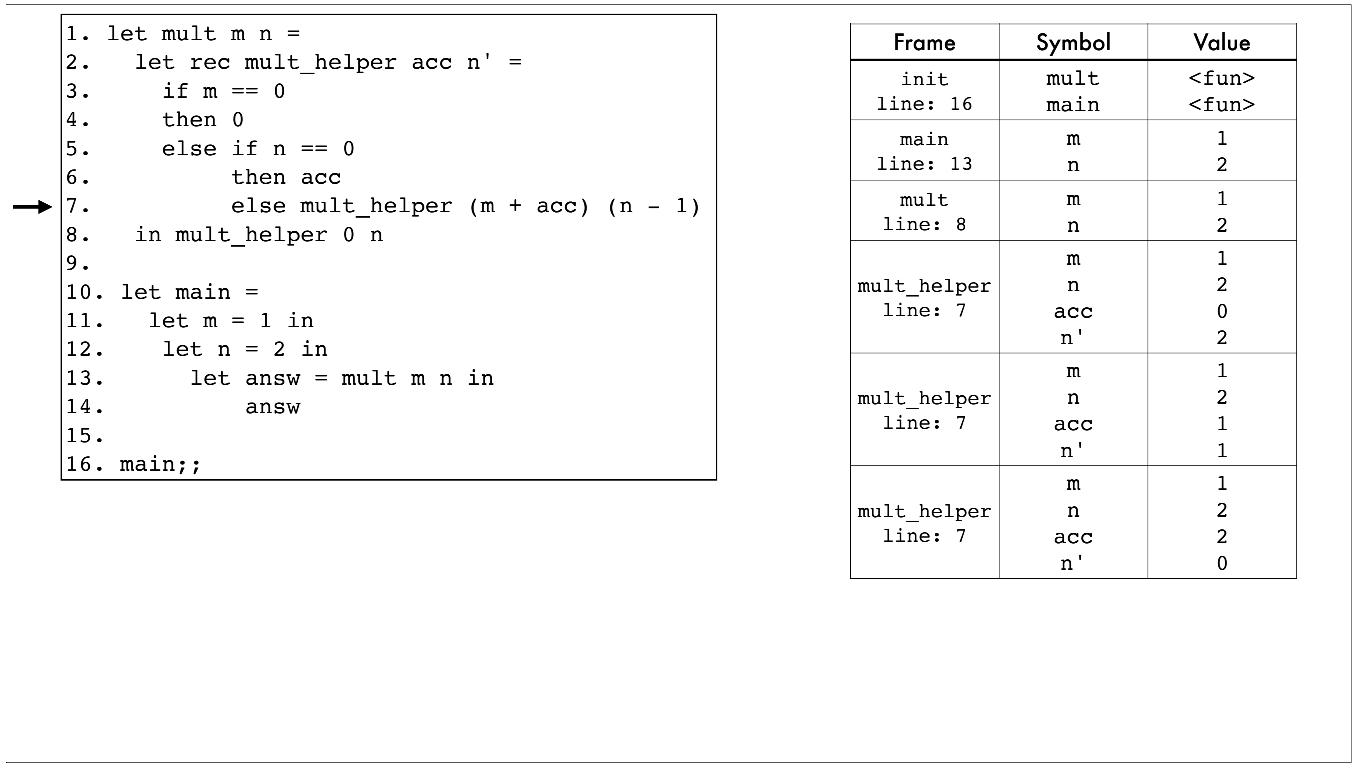
Given the inputs for acc and n' we must now evaluate line 7, but notice that all our bindings are completely filled in. There is no need to update any activation records! This is important, because we will be able to optimize our evaluation, more in a min. Since n' is 2 we must recurse again on line 7...



```
1. let mult m n =
2.   let rec mult_helper acc n' =
3.     if m == 0
4.     then 0
5.     else if n == 0
6.          then acc
7.          else mult_helper (m + acc) (n - 1)
8.   in mult_helper 0 n
9.
10. let main =
11.   let m = 1 in
12.    let n = 2 in
13.      let answ = mult m n in
14.          answ
15.
16. main;;
```

Frame	Symbol	Value
init line: 16	mult	<fun>
	main	<fun>
main line: 13	m	1
	n	2
mult line: 8	m	1
	n	2
mult_helper line: 7	m	1
	n	2
	acc	0
	n'	2
mult_helper line: 7	m	1
	n	2
	acc	1
	n'	1

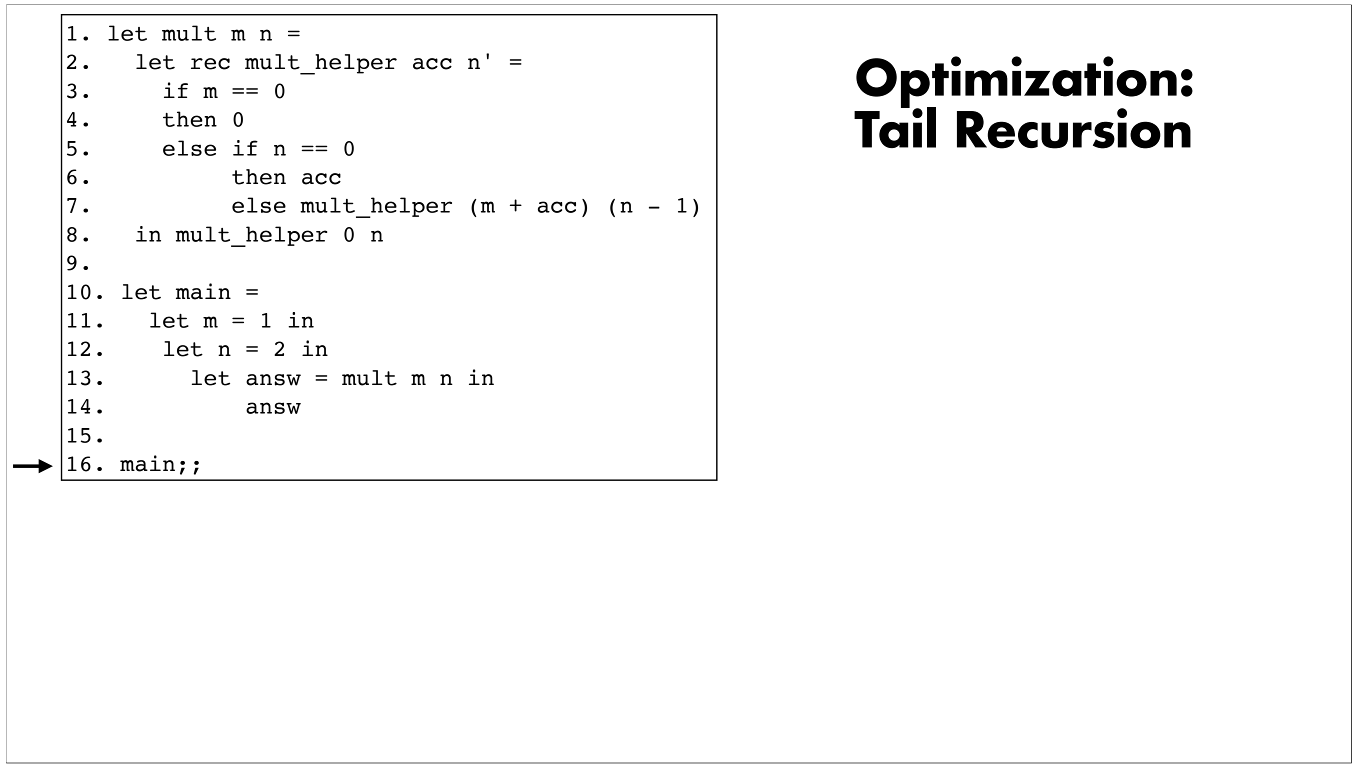
Again, all of our bindings are completely computed. We can see that our accumulator has been updated to 1, and we decrease n' by 1. But, since n' is not 0, we recurse on line 7 again...



```
1. let mult m n =
2.   let rec mult_helper acc n' =
3.     if m == 0
4.     then 0
5.     else if n == 0
6.          then acc
7.          else mult_helper (m + acc) (n - 1)
8.   in mult_helper 0 n
9.
10. let main =
11.   let m = 1 in
12.    let n = 2 in
13.      let answ = mult m n in
14.          answ
15.
16. main;;
```

Frame	Symbol	Value
init line: 16	mult	<fun>
	main	<fun>
main line: 13	m	1
	n	2
mult line: 8	m	1
	n	2
mult_helper line: 7	m	1
	n	2
	acc	0
	n'	2
mult_helper line: 7	m	1
	n	2
	acc	1
	n'	1
mult_helper line: 7	m	1
	n	2
	acc	2
	n'	0

At this point, we can see that the accumulator, acc, is 2, which is our return value, and n' is zero. So we evaluate line 6, and simply return acc. Notice that we do not need to update any activation records, and as a result, we do not need to keep around any of these records except for the last one! This means, the compiler could detect tail recursion, and implement an optimization on its stack usage, let's see how this works, by evaluating the same function again, but in an optimized fashion...



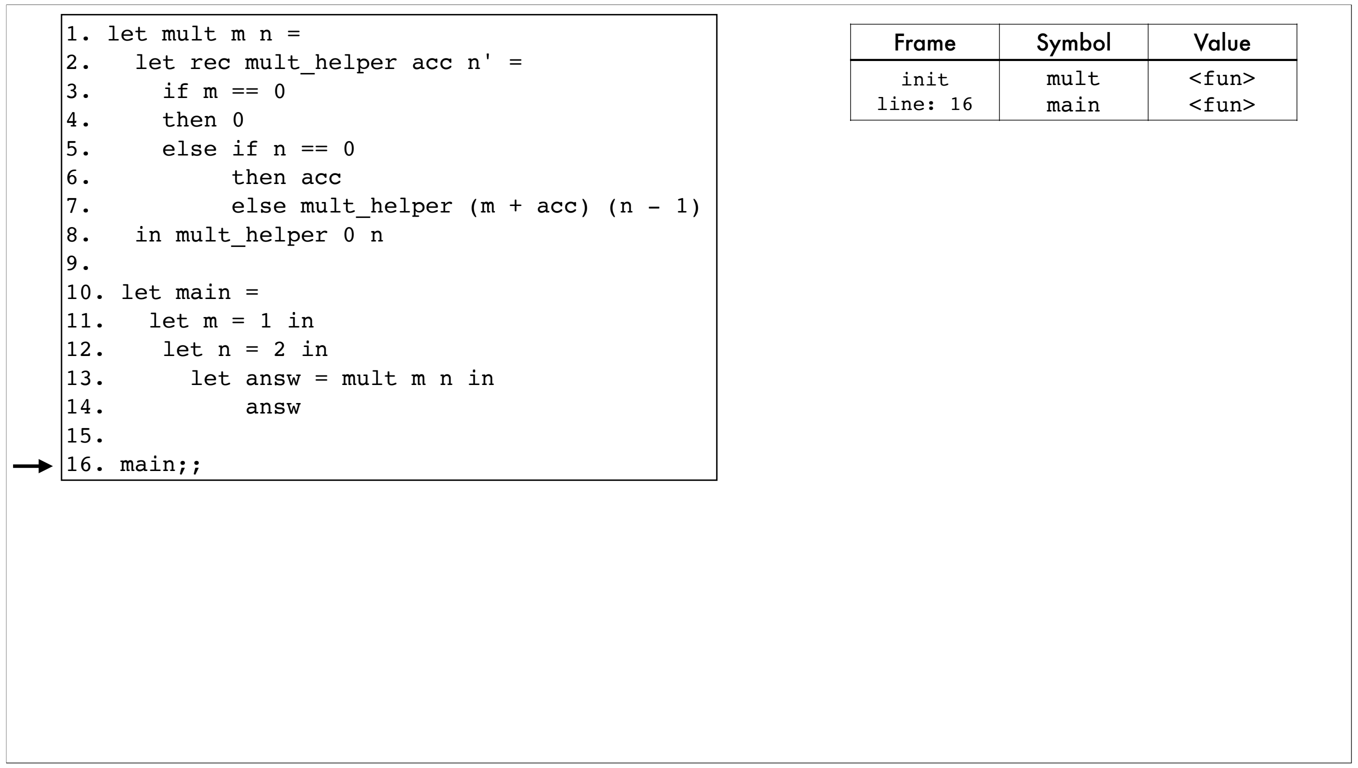
Optimization: Tail Recursion

```
let mult m n =
  let rec mult_helper acc n' =
    if m == 0
    then 0
    else if n == 0
         then acc
         else mult_helper (m + acc) (n - 1)
  in mult_helper 0 n

let main =
  let m = 1 in
   let n = 2 in
     let answ = mult m n in
         answ

main;;
```

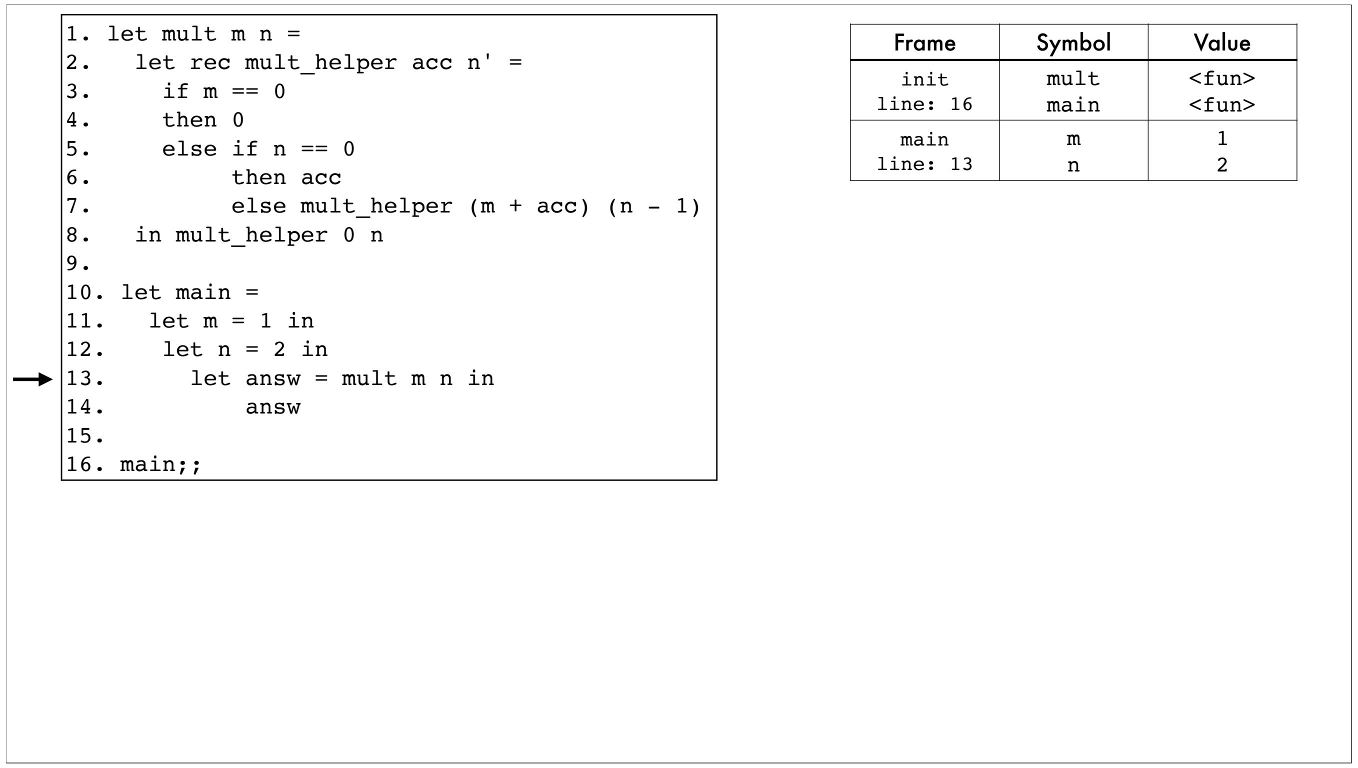
We begin just as we have been... But, now instead of letting the stack grow, we are going to pop at every step!



```
→ 1. let mult m n =
  2.   let rec mult_helper acc n' =
  3.     if m == 0
  4.     then 0
  5.     else if n == 0
  6.          then acc
  7.          else mult_helper (m + acc) (n - 1)
  8.   in mult_helper 0 n
  9.
  10. let main =
  11.   let m = 1 in
  12.    let n = 2 in
  13.      let answ = mult m n in
  14.          answ
  15.
→ 16. main;;
```

Frame	Symbol	Value
init	mult	<fun>
line: 16	main	<fun>

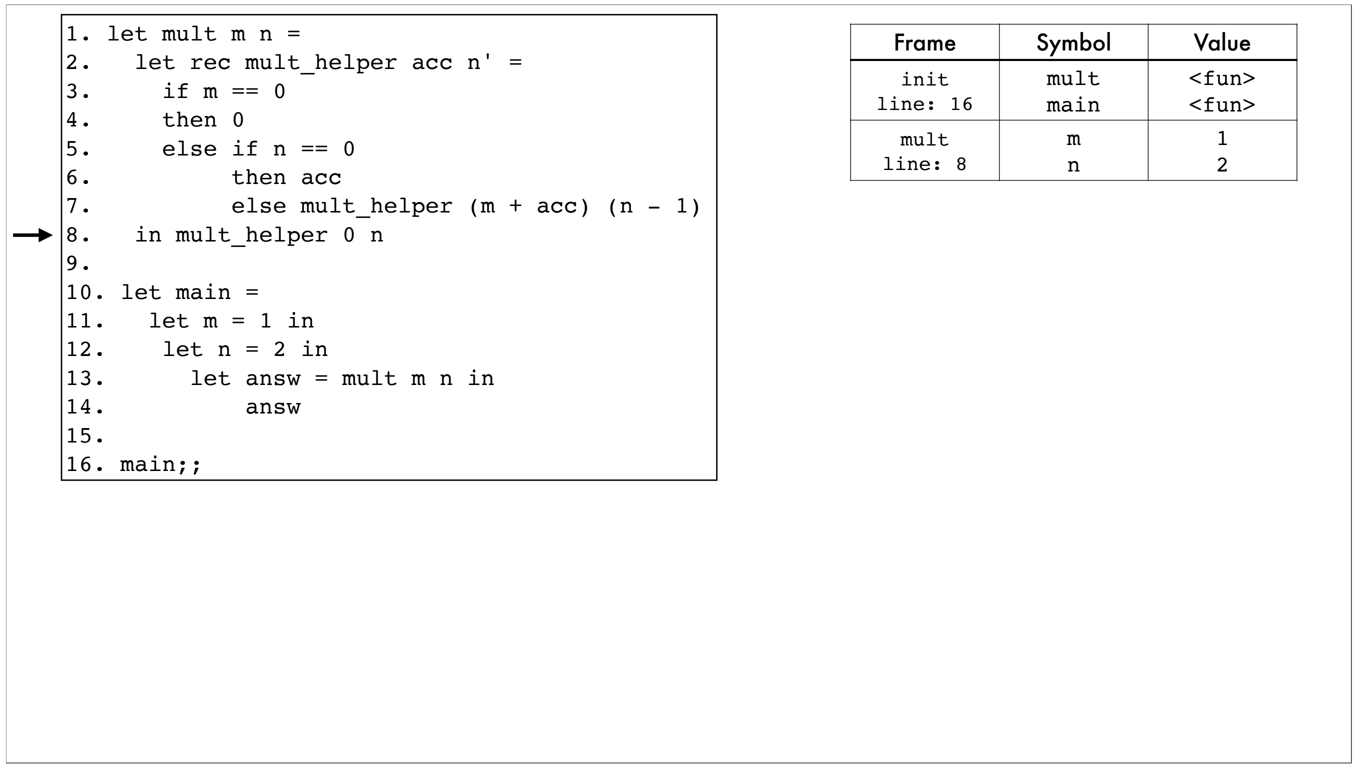
We begin just as we have been... But, now instead of letting the stack grow, we are going to pop at every step!



```
   1. let mult m n =
   2.   let rec mult_helper acc n' =
   3.     if m == 0
   4.     then 0
   5.     else if n == 0
   6.          then acc
   7.          else mult_helper (m + acc) (n - 1)
   8.   in mult_helper 0 n
   9.
   10. let main =
   11.   let m = 1 in
   12.    let n = 2 in
→  13.      let answ = mult m n in
   14.          answ
   15.
   16. main;;
```

Frame	Symbol	Value
init line: 16	mult main	<fun> <fun>
main line: 13	m n	1 2

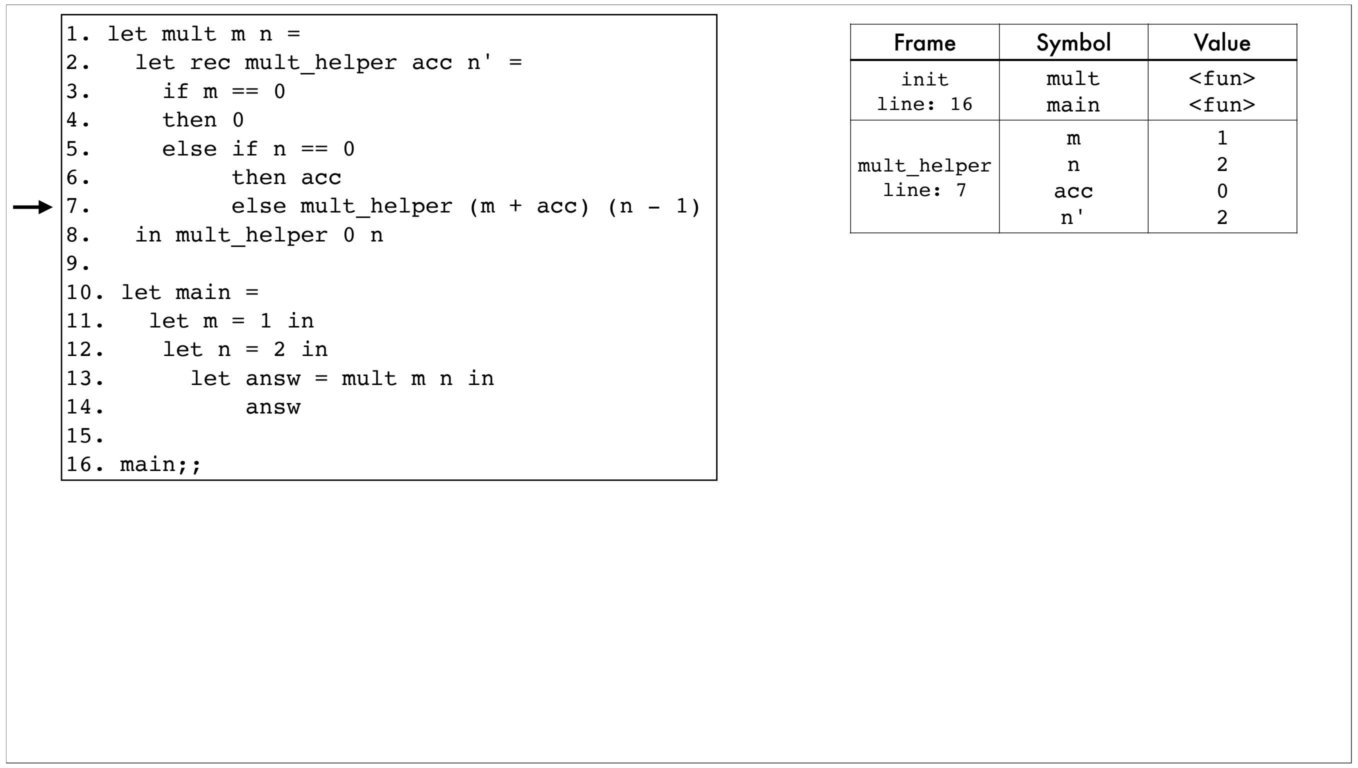
So we have our first function call, but it's return value is the same return value as the next function call, so we pop, and then push...



```
   1. let mult m n =
   2.   let rec mult_helper acc n' =
   3.     if m == 0
   4.     then 0
   5.     else if n == 0
   6.          then acc
   7.          else mult_helper (m + acc) (n - 1)
→  8.   in mult_helper 0 n
   9.
   10. let main =
   11.   let m = 1 in
   12.    let n = 2 in
   13.      let answ = mult m n in
   14.          answ
   15.
   16. main;;
```

Frame	Symbol	Value
init	mult	<fun>
line: 16	main	<fun>
mult	m	1
line: 8	n	2

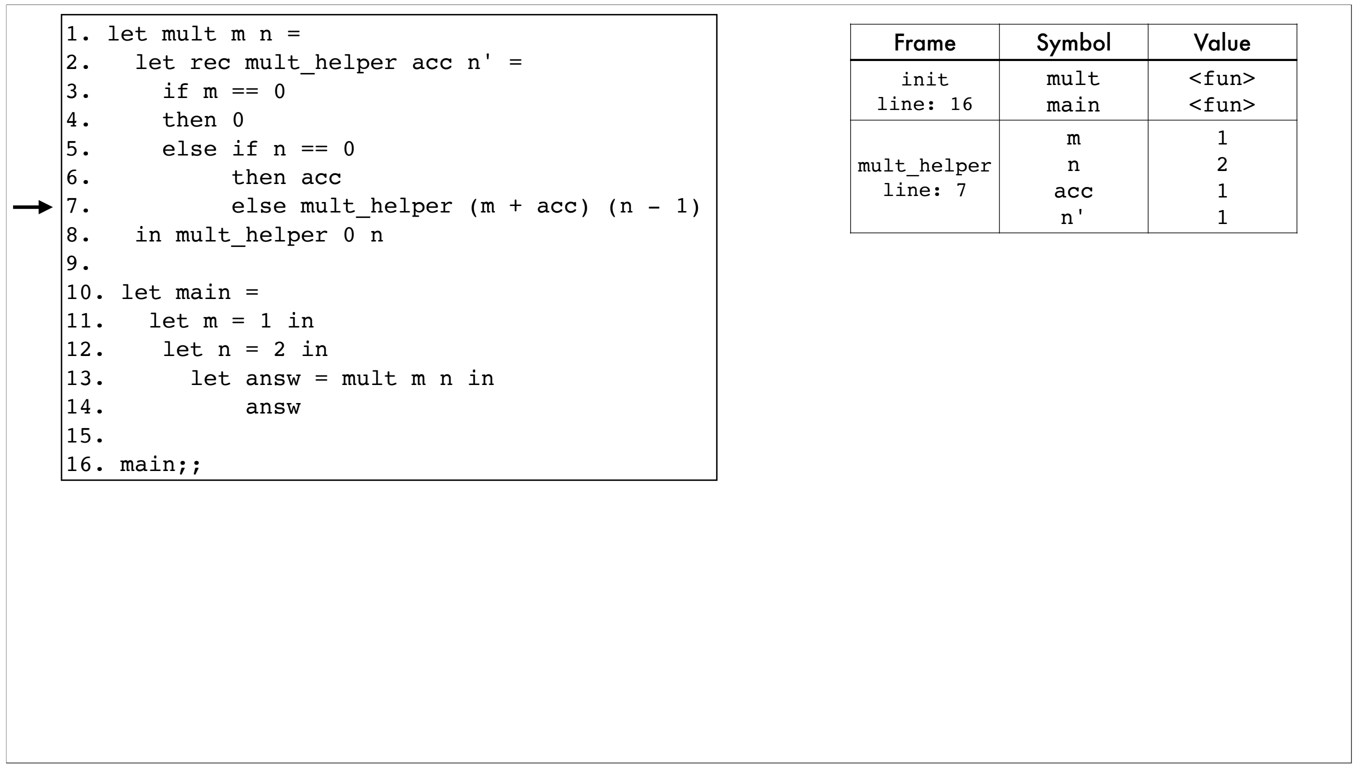
Then, again, pop and push our call to mult_helper on line 8...



```
→  1. let mult m n =
   2.   let rec mult_helper acc n' =
   3.     if m == 0
   4.     then 0
   5.     else if n == 0
   6.          then acc
→  7.          else mult_helper (m + acc) (n - 1)
   8.   in mult_helper 0 n
   9.
  10. let main =
  11.   let m = 1 in
  12.    let n = 2 in
  13.      let answ = mult m n in
  14.          answ
  15.
  16. main;;
```

Frame	Symbol	Value
init line: 16	mult	<fun>
	main	<fun>
mult_helper line: 7	m	1
	n	2
	acc	0
	n'	2

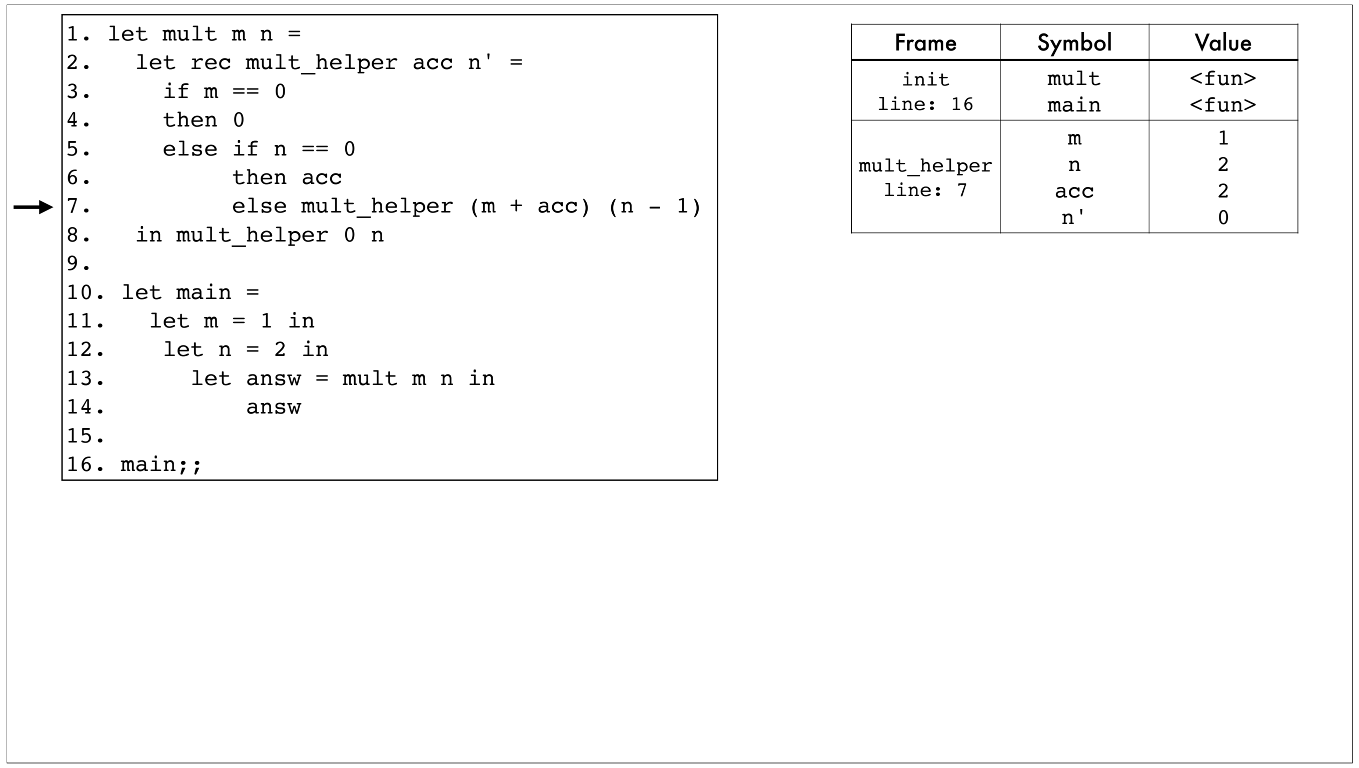
Here our final return value is the same return value as this call to mult_helper! So we can simply pop it, and keep evaluating...



```
   1. let mult m n =
   2.   let rec mult_helper acc n' =
   3.     if m == 0
   4.     then 0
   5.     else if n == 0
   6.          then acc
→  7.          else mult_helper (m + acc) (n - 1)
   8.   in mult_helper 0 n
   9.
   10. let main =
   11.   let m = 1 in
   12.    let n = 2 in
   13.      let answ = mult m n in
   14.          answ
   15.
   16. main;;
```

Frame	Symbol	Value
init line: 16	mult main	<fun> <fun>
mult_helper line: 7	m n acc n'	1 2 1 1

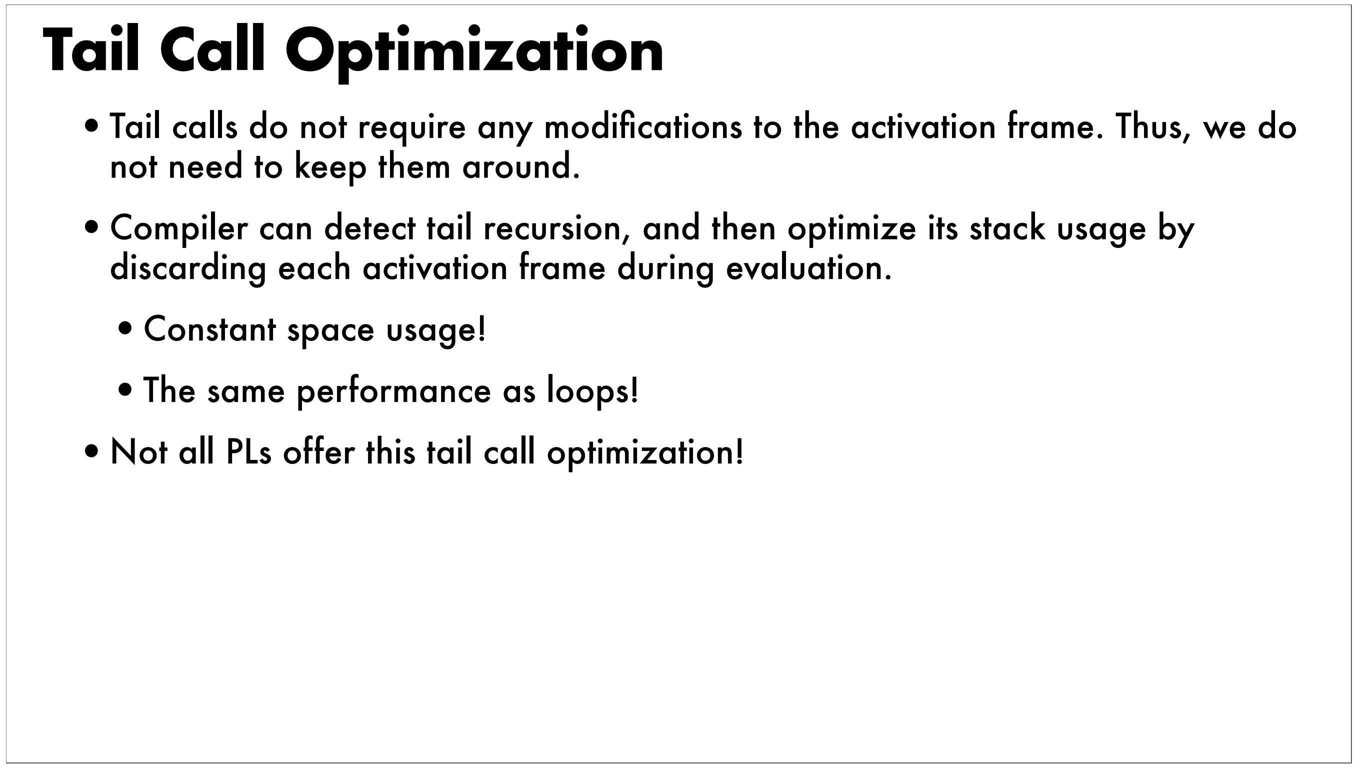
We next have to keep evaluating mult_helper in order to compute acc. So we have one more recursive call....



```
1. let mult m n =
2.   let rec mult_helper acc n' =
3.     if m == 0
4.     then 0
5.     else if n == 0
6.          then acc
7.          else mult_helper (m + acc) (n - 1)
8.   in mult_helper 0 n
9.
10. let main =
11.   let m = 1 in
12.    let n = 2 in
13.      let answ = mult m n in
14.          answ
15.
16. main;;
```

Frame	Symbol	Value
init line: 16	mult main	<fun> <fun>
mult_helper line: 7	m n acc n'	1 2 2 0

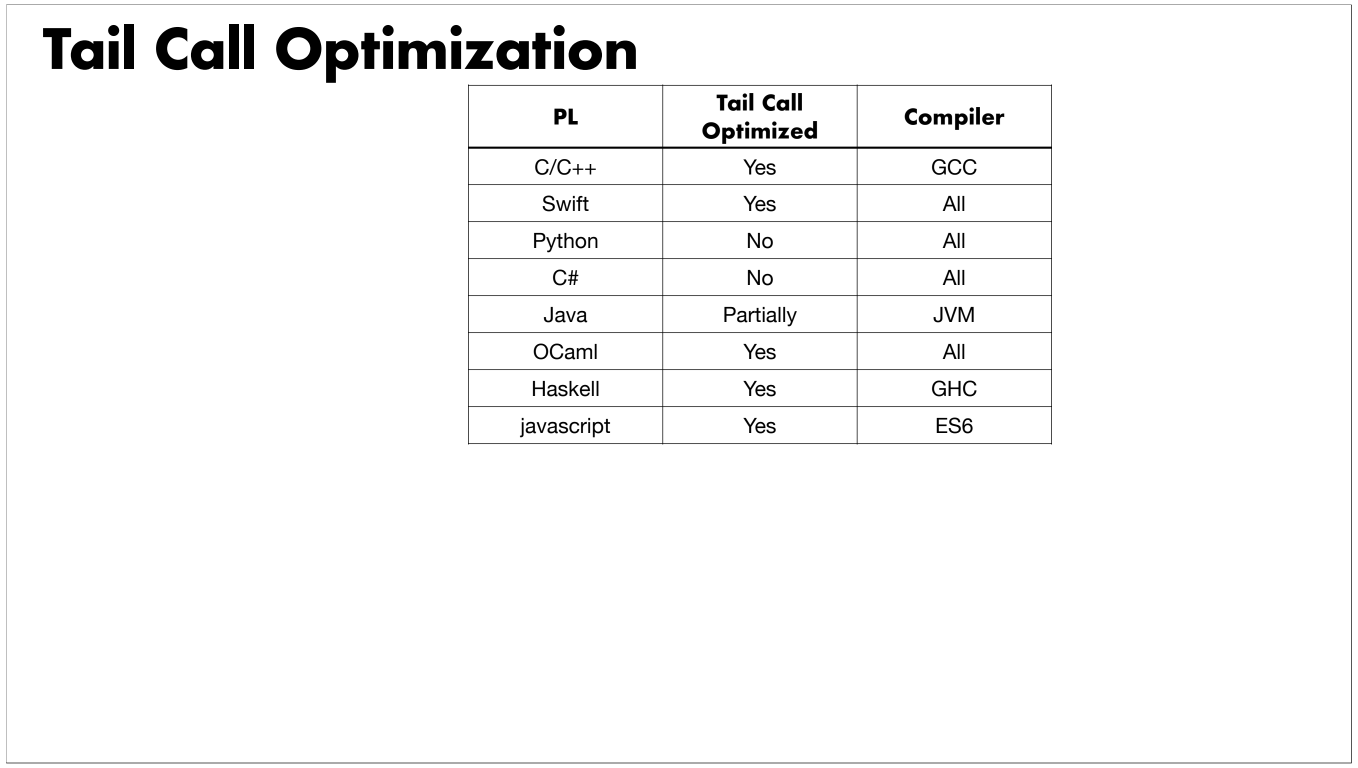
At this point, we are done, and we used a constant amount of stack space to compute this! Thus, a huge performance gain!!



Tail Call Optimization

- Tail calls do not require any modifications to the activation frame. Thus, we do not need to keep them around.
- Compiler can detect tail recursion, and then optimize its stack usage by discarding each activation frame during evaluation.
  - Constant space usage!
  - The same performance as loops!
- Not all PLs offer this tail call optimization!

So let's recap. read slide...



Tail Call Optimization

PL	Tail Call Optimized	Compiler
C/C++	Yes	GCC
Swift	Yes	All
Python	No	All
C#	No	All
Java	Partially	JVM
OCaml	Yes	All
Haskell	Yes	GHC
javascript	Yes	ES6

As we can see when we use recursion, tail recursion can indeed be optimized, but not all programming languages give us this. In the next lecture, we will discuss why we may want to use recursion even when we have loops available, and vice versa.