What is tail-recursion?

Whilst starting to learn lisp, I've come across the term *tail-recursive*. What does it mean?

algorithm language-agnostic functional-programming recursion tail-recursion

edited Oct 8 '08 at 20:29

community wiki
7 revs, 4 users 100%
Ben Lever

25 For curious: both while and whilst have been in the language for a very long time. While was in use in Old English; whilst is a Middle English development of while. As conjunctions they are interchangeable in meaning, but whilst has not survived in standard American English. – Filip Bartuzi Oct 18 '14 at 23:02

16 Answers

Tail recursion is well-described in previous answers, but I think an example in action would help to illustrate the concept.

Consider a simple function that adds the first N integers. (e.g. `sum(5) = 1 + 2 + 3 + 4 + 5 = 15`).

Here is a simple Python implementation that uses recursion:

```
def recsum(x):
 if x == 1:
  return x
 else:
  return x + recsum(x - 1)
```

If you called `recsum(5)`, this is what the Python interpreter would evaluate.

```
recsum(5)
5 + recsum(4)
5 + (4 + recsum(3))
5 + (4 + (3 + recsum(2)))
5 + (4 + (3 + (2 + recsum(1))))
5 + (4 + (3 + (2 + 1)))
15
```

Note how every recursive call has to complete before the Python interpreter begins to actually do the work of calculating the sum.

Here's a tail-recursive version of the same function:

```
def tailrecsum(x, running_total=0):
  if x == 0:
    return running_total
  else:
    return tailrecsum(x - 1, running_total + x)
```

Here's the sequence of events that would occur if you called `tailrecsum(5)`, (which would effectively be `tailrecsum(5, 0)`, because of the default second argument).

```
tailrecsum(5, 0)
tailrecsum(4, 5)
tailrecsum(3, 9)
tailrecsum(2, 12)
tailrecsum(1, 14)
tailrecsum(0, 15)
15
```

In the tail-recursive case, with each evaluation of the recursive call, the `running_total` is updated.

Note: As mentioned in the comments, Python doesn't have built-in support for optimizing away tail calls, so there's no advantage to doing this in Python. However, you can use a decorator to achieve the optimization.

edited Aug 12 '14 at 23:08
Mr. Polywhirl
6,415 ● 4 ● 16 ● 41

answered Aug 31 '08 at 18:21
Lorin Hochstein
16.4k ● 11 ● 56 ● 90

38 Python is kind of an odd choice here, since it does not have AFAIK tail-recursion elimination. – Chris Conway Sep 21 '08 at 14:43

1 Chris Conway is correct. Tail calls are not optimized in Python, unfortunately. Guido claims having the stack available for debugging is better than TCO. – McPherrinM Dec 5 '09 at 3:25

21 You'll find Guido's opinion here. – new123456 May 14 '11 at 14:27

2 @Paco, what was the link? Was it some kind of a joke? It is 404, do you have a mirror? :-) – tillda Aug 29 '12 at 23:41

51 @tillda: Yes a joke. Here is a mirror: cs.cmu.edu/~wklieber/python-tail-recursion.jpg – Paco Sep 18 '12 at 14:34

In **traditional recursion**, the typical model is that you perform your recursive calls first, and then you take the return value of the recursive call and calculate the result. In this manner, you don't get the result of your calculation until you have returned from every recursive call.

In **tail recursion**, you perform your calculations first, and then you execute the recursive call, passing the results of your current step to the next recursive step. This results in the last statement being in the form of "(return (recursive-function params))" (I think that's the syntax for Lisp). **Basically, the return value of any given recursive step is the same as the return value of the next recursive call**.

The consequence of this is that once you are ready to perform your next recursive step, you don't need the current stack frame any more. This allows for some optimization. In fact, with an appropriately written compiler, you should never have a stack overflow *snicker* with a tail recursive call. Simply reuse the current stack frame for the next recursive step. I'm pretty sure Lisp does this.

edited May 8 at 17:19
haccks
50k ● 9 ● 55 ● 110

answered Aug 29 '08 at 3:57
Daniel F. Hanson
3,948 ● 1 ● 12 ● 15

4 "I'm pretty sure Lisp does this" -- Scheme does, but Common Lisp doesn't always. – Aaron Jan 2 '09 at 23:51

1 @Daniel "Basically, the return value of any given recursive step is the same as the return value of the next recursive call."- I fail to see this argument holding true for the code snippet posted by Lorin Hochstein. Can you please elaborate? – Geek Mar 20 '13 at 19:16

3 @Geek This is a really late response, but that is actually true in Lorin Hochstein's example. The calculation for each step is done before the recursive call, rather than after it. As a result, each stop just returns the value directly from the previous step. The last recursive call finishes the computation and then returns the final result unmodified all the way back down the call stack. – reirab Apr 23 '14 at 22:58

Scala does but you need the @tailrec specified to enforce it. – SilentDirge Dec 26 '14 at 21:02

"In this manner, you don't get the result of your calculation until you have returned from every recursive call." -- maybe I misunderstood this, but this isn't particularly true for lazy languages where the *traditional recursion* is the only way to actually get a result without calling all recursions (e.g. folding over an infinite list of Bools with &&). – hasufell Jun 30 at 21:12

An important point is that tail recursion is essentially equivalent to looping. It's not just a matter of compiler optimization, but a fundamental fact about expressiveness. This goes both ways: you can take any loop of the form

```
while(E) { S }; return Q
```

where `E` and `Q` are expressions and `S` is a sequence of statements, and turn it into a tail recursive function

```
f() = if E then { S; return f() } else { return Q }
```

Of course, `E`, `S`, and `Q` have to be defined to compute some interesting value over some variables. For example, the looping function

```
sum(n) {
  int i = 1, k = 0;
  while( i <= n ) {
    k += i;
    ++i;
  }
  return k;
}
```

is equivalent to the tail-recursive function(s)

```
sum_aux(n,i,k) {
  if( i <= n ) {
    return sum_aux(n,i+1,k+i);
  } else {
    return k;
  }
}

sum(n) {
  return sum_aux(n,1,0);
}
```

(This "wrapping" of the tail-recursive function with a function with fewer parameters is a common functional idiom.)

edited Aug 12 '14 at 23:28

Mr. Polywhirl
6,415 • 4 • 16 • 41

answered Aug 31 '08 at 17:29

Chris Conway
28.2k • 22 • 92 • 135

5 Great explanation, but why is your function called fibo? That's not Fibonacci; it's f(n) = Binomial(n+1,2) – RexE Dec 25 '11 at 7:29

5 @RexE Total brain fart. I got upvoted 20 times and nobody pointed that out! – Chris Conway Dec 26 '11 at 14:24

1 :) also make sure to rename the recursive calls. – RexE Dec 28 '11 at 20:45

In the answer by @LorinHochstein I understood, based on his explanation, that tail recursion to be when the recursive portion follows "Return", however in yours, the tail recursive is not. Are you sure your example is properly considered tail recursion? – Imray Mar 10 '13 at 22:44

@Imray The tail-recursive part is the "return sum_aux" statement inside sum_aux. – Chris Conway Mar 11 '13 at 4:51

This excerpt from the book *Programming in Lua* shows how to make a proper tail recursion (in Lua, but should apply to Lisp too) and why it's better.

> A tail call [tail recursion] is a kind of goto dressed as a call. A tail call happens when a function calls another as its last action, so it has nothing else to do. For instance, in the following code, the call to g is a tail call:

```
function f (x)
  return g(x)
end
```

> After f calls g, it has nothing else to do. In such situations, the program does not need to return to the calling function when the called function ends. Therefore, after the tail call, the program does not need to keep any information about the calling function in the stack.
>
> Because a proper tail call uses no stack space, there is no limit on the number of "nested" tail calls that a program can make. For instance, we can call the following function with any number as argument; it will never overflow the stack:

```
function foo (n)
  if n > 0 then return foo(n - 1) end
```

```
end
```

As I said earlier, a tail call is a kind of goto. As such, a quite useful application of proper tail calls in Lua is for programming state machines. Such applications can represent each state by a function; to change state is to go to (or to call) a specific function. As an example, let us consider a simple maze game. The maze has several rooms, each with up to four doors: north, south, east, and west. At each step, the user enters a movement direction. If there is a door in that direction, the user goes to the corresponding room; otherwise, the program prints a warning. The goal is to go from an initial room to a final room.

This game is a typical state machine, where the current room is the state. We can implement such maze with one function for each room. We use tail calls to move from one room to another. A small maze with four rooms could look like this:

```
function room1 ()
  local move = io.read()
  if move == "south" then return room3()
  elseif move == "east" then return room2()
  else print("invalid move")
       return room1()   -- stay in the same room
  end
end

function room2 ()
  local move = io.read()
  if move == "south" then return room4()
  elseif move == "west" then return room1()
  else print("invalid move")
       return room2()
  end
end

function room3 ()
  local move = io.read()
  if move == "north" then return room1()
  elseif move == "east" then return room4()
  else print("invalid move")
       return room3()
  end
end

function room4 ()
  print("congratulations!")
end
```

So you see, when you make a recursive call like:

```
function x(n)
  if n==0 then return 0
  n= n-2
  return x(n) + 1
end
```

This is not tail recursive because you still have things to do (add 1) in that function after the recursive call is made. If you input a very high number it will probably cause a stack overflow.

edited Aug 12 '14 at 23:20

Mr. Polywhirl
6,415 ● 4 ● 16 ● 41

answered Aug 29 '08 at 16:03

Hoffmann
4,470 ● 5 ● 36 ● 52

The link is broken. It should be lua.org/pil/6.3.html – Alexander Gladysh Jun 18 '10 at 2:40

This is a great answer because it explains the implications of tail calls upon stack size. – Andrew Swan Aug 22 '14 at 7:32

@AndrewSwan Indeed, although I believe that the original asker and the occasional reader who might stumble into this question might be better served with the accepted answer (since he might not know what the stack actually is.) By the way I use Jira, big fan. – Hoffmann Aug 22 '14 at 13:47

Instead of explaining it with words, here's an example. This is a Scheme version of the factorial function:

```
(define (factorial x)
  (if (= x 0) 1
      (* x (factorial (- x 1)))))
```

Here is a version of factorial that is tail-recursive:

```
(define factorial
  (letrec ((fact (lambda (x accum)
                   (if (= x 0) accum
                       (fact (- x 1) (* accum x))))))
    (lambda (x)
      (fact x 1))))
```

You will notice in the first version that the recursive call to fact is fed into the multiplication expression, and therefore the state has to be saved on the stack when making the recursive call. In the tail-recursive version there is no other S-expression waiting for the value of the recursive call, and since there is no further work to do, the state doesn't have to be saved on the stack. As a rule, Scheme tail-recursive functions use constant stack space.

edited Apr 22 '13 at 22:32

Cristian Ciupitu
7,341 ● 3 ● 24 ● 41

answered Aug 29 '08 at 3:57

Kyle Cronin
40.1k ● 31 ● 118 ● 143

Using regular recursion, each recursive call pushes another entry onto the call stack. When the recursion is completed, the app then has to pop each entry off all the way back down.

With tail recursion, the compiler is able to collapse the stack down to one entry, so you save stack space...A large recursive query can actually cause a stack overflow.

Basically Tail recursions are able to be optimized into iteration.

answered Aug 29 '08 at 3:55

FlySwat
72.5k ● 41 ● 198 ● 284

The jargon file has this to say about the definition of tail recursion:

tail recursion /n./

If you aren't sick of it already, see tail recursion.

edited Feb 16 '10 at 17:34

answered Aug 29 '08 at 7:21

Pat
21.1k ● 11 ● 55 ● 81

It means that rather than needing to push the instruction pointer on the stack, you can simply jump to the top of a recursive function and continue execution. This allows for functions to recurse indefinitely without overflowing the stack.

I wrote a blog post on the subject, which has graphical examples of what the stack frames look like.

answered Aug 31 '08 at 23:52

Chris Smith
8,220 ● 6 ● 36 ● 62

Neat article, thanks! – vtortola Nov 12 '14 at 12:45

Tail recursion refers to the recursive call being last in the last logic instruction in the recursive algorithm.

Typically in recursion you have a *base-case* which is what stops the recursive calls and begins popping the call stack. To use a classic example, though more C-ish than Lisp, the factorial function illustrates tail recursion. The recursive call occurs *after* checking the base-case condition.

```
factorial(x, fac) {
  if (x == 1)
     return fac;
   else
     return factorial(x-1, x*fac);
}
```

Note, the initial call to factorial must be factorial(n, 1) where n is the number for which the factorial is to be calculated.

edited Aug 12 '14 at 23:30

Mr. Polywhirl
6,415 ● 4 ● 16 ● 41

answered Aug 29 '08 at 3:57

Peter Meyer
13.5k ● 1 ● 22 ● 47

1 Your function does not have tail recursion. – leppie Oct 1 '08 at 22:21

1 My bad. Terrible oversight! I have corrected the algorithm. Thanks for the comment on something a month old! – Peter Meyer Oct 2 '08 at 2:02

In Java, here's a possible tail recursive implementation of the Fibonacci function:

```
public int tailRecursive(final int n) {
    if (n <= 2)
        return 1;
    return tailRecursiveAux(n, 1, 1);
}

private int tailRecursiveAux(int n, int iter, int acc) {
    if (iter == n)
        return acc;
    return tailRecursiveAux(n, ++iter, acc + iter);
}
```

Contrast this with the standard recursive implementation:

```
public int recursive(final int n) {
    if (n <= 2)
        return 1;
    return recursive(n - 1) + recursive(n - 2);
}
```

edited Aug 12 '14 at 23:24

Mr. Polywhirl
6,415 ● 4 ● 16 ● 41

answered Oct 14 '08 at 21:20

jorgetown
195 ● 9

This is returning wrong results for me, for input 8 I get 36, it has to be 21. Am I missing something? I'm using java and copy pasted it. – Alberto Zaccagni Nov 28 '11 at 21:21

1 This returns SUM(i) for i in [1, n]. Nothing to do with Fibbonacci. For a Fibbo, you need a tests which substracts `iter` to `acc` when `iter < (n-1)`. – Askolein Mar 15 '13 at 13:44

Here is a quick code snippet comparing two functions. The first is traditional recursion for finding the factorial of a given number. The second uses tail recursion.

Very simple and intuitive to understand.

Easy way to tell if a recursive function is tail recursive, is if it returns a concrete value in the base case. Meaning that it doesn't return 1 or true or anything like that. It will more then likely return some variant of one of the method paramters.

Another way is to tell is if the recursive call is free of any addition, arithmetic, modification, etc... Meaning its nothing but a pure recursive call.

```
public static int factorial(int mynumber) {
    if (mynumber == 1) {
        return 1;
    } else {
        return mynumber * factorial(--mynumber);
    }
}

public static int tail_factorial(int mynumber, int sofar) {
    if (mynumber == 1) {
        return sofar;
    } else {
        return tail_factorial(--mynumber, sofar * mynumber);
    }
}
```

edited Aug 12 '14 at 23:25

community wiki
2 revs, 2 users 88%
AbuZubair

1 0! is 1. So "mynumber == 1" should be "mynumber == 0". – polerto Mar 5 '14 at 23:35

I'm not a Lisp programmer, but I think this will help.

Basically it's a style of programming such that the recursive call is the last thing you do.

answered Aug 29 '08 at 3:50

Matt Hamilton
113k ● 37 ● 281 ● 275

Here is a Common Lisp example that does factorials using tail-recursion. Due to the stack-less nature, one could perform insanely large factorial computations ...

```
(defun ! (n &optional (product 1))
    (if (zerop n) product
        (! (1- n) (* product n))))
```

And then for fun you could try `(format nil "~R" (! 25))`

edited Aug 12 '14 at 23:27

community wiki
3 revs, 2 users 71%
user922475

here is a Perl 5 version of the `tailrecsum` function mentioned earlier.

```
sub tail_rec_sum($;$){
  my( $x,$running_total ) = (@_,0);

  return $running_total unless $x;

  @_ = ($x-1,$running_total+$x);
  goto &tail_rec_sum; # throw away current stack frame
}
```

edited Aug 12 '14 at 23:29

Mr. Polywhirl
6,415 ● 4 ● 16 ● 41

answered Oct 1 '08 at 22:06

Brad Gilbert
13.6k ● 4 ● 42 ● 76

To understand some of the core differences between tail-call recursion and non-tail-call recursion we can explore the .NET implementations of these techniques.

Here is an article with some examples in C#, F#, and C++\CLI: Adventures in Tail Recursion in C#, F#, and C++\CLI.

C# does not optimize for tail-call recursion whereas F# does.

The differences of principle involve loops vs. Lambda calculus. C# is designed with loops in mind whereas F# is built from the principles of Lambda calculus. For a very good (and free) book on the principles of Lambda calculus, see: Structure and Interpretation of Computer Programs, by Abelson, Sussman, and Sussman.

Regarding tail calls in F#, for a very good introductory article , see: Detailed Introduction to Tail Calls in F#. Finally, here is an article that covers the difference between non-tail recursion and tail-call recursion (in F#): Tail-recursion vs. non-tail recursion in F sharp.

If you want to read about some of the design differences of tail-call recursion between C# and F#, see: Generating Tail-Call Opcode in C# and F#.

If you care enough to want to know what conditions prevent the C# compiler from performing tail-call optimizations, see this article: JIT CLR tail-call conditions.

answered Apr 28 '14 at 19:13

community wiki
bostIT

Recursion means a function calling itself. For example:

```
(define (un-ended name)
  (un-ended 'me)
  (print "How can I get here?"))
```

Tail-Recursion means the recursion that conclude the function:

```
(define (un-ended name)
  (print "hello")
  (un-ended 'me))
```

See, the last thing un-ended function (procedure, in Scheme jargon) does is to call itself. Another (more useful) example is:

```
(define (map lst op)
  (define (helper done left)
    (if (nil? left)
        done
        (helper (cons (op (car left))
                      done)
                (cdr left))))
  (reverse (helper '() lst)))
```

In the helper procedure, the LAST thing it does if left is not nil is to call itself (AFTER cons something and cdr something). This is basically how you map a list.

The tail-recursion has a great advantage that the interperter (or compiler, dependent on the language and vendor) can optimize it, and transform it into something equivalent to a while loop. As matter of fact, in Scheme tradition, most "for" and "while" loop is done in tail-recursion manner (there is no for and while, as far as I know).

edited Aug 12 '14 at 23:22

Mr. Polywhirl
6,415 ● 4 ● 16 ● 41

answered Sep 2 '08 at 14:08

magice
87 ● 3

protected by Srikar Appal Aug 4 '13 at 15:27

Thank you for your interest in this question. Because it has attracted low-quality answers, posting an answer now requires 10 reputation on this site.

Would you like to answer one of these unanswered questions instead?