WIKIPEDIA

Tail call

In computer science, a **tail call** is a subroutine call performed as the final action of a procedure.[1] If a tail call might lead to the same subroutine being called again later in the call chain, the subroutine is said to be **tail-recursive**, which is a special case of recursion. **Tail recursion** (or **tail-end recursion**) is particularly useful, and often easy to handle in implementations.

Tail calls can be implemented without adding a new stack frame to the call stack. Most of the frame of the current procedure is no longer needed, and can be replaced by the frame of the tail call, modified as appropriate (similar to overlay for processes, but for function calls). The program can then jump to the called subroutine. Producing such code instead of a standard call sequence is called **tail call elimination**. Tail call elimination allows procedure calls in tail position to be implemented as efficiently as goto statements, thus allowing efficient structured programming. In the words of Guy L. Steele, "in general, procedure calls may be usefully thought of as GOTO statements which also pass parameters, and can be uniformly coded as [machine code] JUMP instructions."[2]

Not all programming languages require tail call elimination. However, in functional programming languages, tail call elimination is often guaranteed by the language standard, allowing tail recursion to use a similar amount of memory as an equivalent loop. The special case of tail recursive calls, when a function calls itself, may be more amenable to call elimination than general tail calls. When the language semantics do not explicitly support general tail calls, a compiler can often still optimize **sibling calls**, or tail calls to functions which take and return the same types as the caller.[3]

Contents

Description

When a function is called, the computer must "remember" the place it was called from, the *return address*, so that it can return to that location with the result once the call is complete. Typically, this information is saved on the call stack, a simple list of return locations in order of the times that the call locations they describe were reached. For tail calls, there is no need to remember the caller – instead, tail call elimination leaves the stack alone (except possibly for function arguments and local variables,[4]) and the newly-called function will return directly to the *original* caller. The tail call doesn't have to appear lexically after all other statements in the source code; it is only important that the calling function return immediately after the tail call, returning the tail call's result if any, since the calling function will never get a chance to do anything after the call if the optimization is performed.

For non-recursive function calls, this is usually an optimization that saves only a little time and space, since there are not that many different functions available to call. When dealing with recursive or mutually recursive functions where recursion happens through tail calls, however, the stack space and the number of returns saved can grow to be very significant, since a function can call itself, directly or indirectly, creating a new call stack frame each time. Tail call elimination often asymptotically reduces stack space requirements from linear, or O(n), to constant, or O(1). Tail call elimination is thus required by the standard definitions of some programming languages, such as Scheme,[5][6] and languages in the ML family among others. The Scheme language definition formalizes the intuitive notion of tail position exactly, by specifying which syntactic forms allow having results in tail context.[7] Implementations allowing an unlimited number of tail calls to be active at the same moment, thanks to tail call elimination, can also be called 'properly tail-recursive'.[5]

Besides space and execution efficiency, tail call elimination is important in the functional programming idiom known as continuation-passing style (CPS), which would otherwise quickly run out of stack space.

Syntactic form

A tail call can be located just before the syntactical end of a function:

```
function foo(data) {
    a(data);
    return b(data);
}
```

Here, both `a(data)` and `b(data)` are calls, but `b` is the last thing the procedure executes before returning and is thus in tail position. However, not all tail calls are necessarily located at the syntactical end of a subroutine:

```
function bar(data) {
    if ( a(data) ) {
        return b(data);
    }
    return c(data);
}
```

Here, both calls to `b` and `c` are in tail position. This is because each of them lies in the end of if-branch respectively, even though the first one is not syntactically at the end of `bar`'s body.

In this code:

```
function foo1(data) {
    return a(data) + 1;
}
```

```
function foo2(data) {
    var ret = a(data);
    return ret;
}
```

```
function foo3(data) {
    var ret = a(data);
    return (ret == 0) ? 1 : ret;
}
```

the call to `a(data)` is in tail position in `foo2`, but it is **not** in tail position either in `foo1` or in `foo3`, because control must return to the caller to allow it to inspect or modify the return value before returning it.

Example programs

The following program is an example in Scheme:[8]

```
;; factorial : number -> number
;; to calculate the product of all positive
;; integers less than or equal to n.
(define (factorial n)
 (if (= n 1)
    1
    (* n (factorial (- n 1)))))
```

This is not written in a tail recursion style, because the multiplication function ("*") is in the tail position. This can be compared to:

```
;; factorial : number -> number
;; to calculate the product of all positive
;; integers less than or equal to n.
```

```
(define (factorial n)
  (fact-iter 1 n))
(define (fact-iter product n)
  (if (< n 2)
      product
      (fact-iter (* product n)
                 (- n 1))))
```

This program assumes applicative-order evaluation. The inner procedure `fact-iter` calls itself *last* in the control flow. This allows an interpreter or compiler to reorganize the execution which would ordinarily look like this:[8]

```
call factorial (4)
 call fact-iter (1 4)
  call fact-iter (4 3)
   call fact-iter (12 2)
    call fact-iter (24 1)
    return 24
   return 24
  return 24
 return 24
return 24
```

into the more efficient variant, in terms of both space and time:

```
call factorial (4)
 call fact-iter (1 4)
 replace arguments with (4 3)
 replace arguments with (12 2)
 replace arguments with (24 1)
 return 24
return 24
```

This reorganization saves space because no state except for the calling function's address needs to be saved, either on the stack or on the heap, and the call stack frame for `fact-iter` is reused for the intermediate results storage. This also means that the programmer need not worry about running out of stack or heap space for extremely deep recursions. In typical implementations, the tail recursive variant will be substantially faster than the other variant, but only by a constant factor.

Some programmers working in functional languages will rewrite recursive code to be tail-recursive so they can take advantage of this feature. This often requires addition of an "accumulator" argument (`product` in the above example) to the function. In some cases (such as filtering lists) and in some languages, full tail recursion may require a function that was previously purely functional to be written such that it mutates references stored in other variables.

Tail recursion modulo cons

Tail recursion modulo cons is a generalization of tail recursion optimization introduced by David H. D. Warren[9] in the context of compilation of Prolog, seen as an *explicitly set once* language. It was described (though not named) by Daniel P. Friedman and David S. Wise in 1974[10] as a LISP compilation technique. As the name suggests, it applies when the

only operation left to perform after a recursive call is to prepend a known value in front of a list returned from it (or to perform a constant number of simple data-constructing operations, in general). This call would thus be a *tail call* save for ("modulo") the said *cons* operation. But prefixing a value at the start of a list *on exit* from a recursive call is the same as appending this value at the end of the growing list *on entry* into the recursive call, thus building the list as a side effect, as if in an implicit accumulator parameter. The following Prolog fragment illustrates the concept:

Prolog example

```
% tail recursive modulo cons:
partition([], _, [], []).
partition([X|Xs], Pivot, [X|Rest], Bigs) :-
  X @< Pivot, !,
  partition(Xs, Pivot, Rest, Bigs).
partition([X|Xs], Pivot, Smalls, [X|Rest])
:-
  partition(Xs, Pivot, Smalls, Rest).
```

```
-- In Haskell, guarded recursion:
partition :: Ord a => [a] -> a -> ([a],[a])
partition [] _ = ([],[])
partition (x:xs) p | x < p     = (x:a,b)
                   | otherwise = (a,x:b)
   where
      (a,b) = partition xs p
```

```
% With explicit unifications:
%     non-tail recursive translation:
partition([], _, [], []).
partition(L, Pivot, Smalls, Bigs) :-
L=[X|Xs],
 (  X @< Pivot
 -> partition(Xs,Pivot,Rest,Bigs), Smalls=
[X|Rest]
 ;  partition(Xs,Pivot,Smalls,Rest),
Bigs=[X|Rest]
 ).
```

```
% With explicit unifications:
%     tail recursive translation:
partition([], _, [], []).
partition(L, Pivot, Smalls, Bigs) :-
L=[X|Xs],
 (  X @< Pivot
 -> Smalls=[X|Rest],
partition(Xs,Pivot,Rest,Bigs)
 ;  Bigs=[X|Rest],
partition(Xs,Pivot,Smalls,Rest)
 ).
```

Thus in tail recursive translation such a call is transformed into first creating a new list node and setting its `first` field, and *then* making a tail call with the pointer to the node's `rest` field as argument, to be filled recursively.

C example

The following fragment defines a recursive function in C that duplicates a linked list:

```
struct list {void *value;struct list *next;};
typedef struct list list;

list *duplicate(const list *ls)
{
    list *head = NULL;

    if (ls != NULL) {
        list *p = duplicate(ls->next);
        head = malloc(sizeof *head);
        head->value = ls->value;
        head->next = p;
    }
```

```
;; in Scheme,
(define (duplicate ls)
  (if (not (null? ls))
    (cons (car ls)
          (duplicate (cdr ls)))
    '()))
```

```
%% in Prolog,
dup([X|Xs],R):-
  dup(Xs,Ys),
```

```
    return head;
}
```

```
  R=[X|Ys].
dup([],[]).
```

In this form the function is not tail-recursive, because control returns to the caller after the recursive call duplicates the rest of the input list. Even if it were to allocate the *head* node before duplicating the rest, it would still need to plug in the result of the recursive call into the `next` field *after* the call.[a] So the function is *almost* tail-recursive. Warren's method pushes the responsibility of filling the `next` field into the recursive call itself, which thus becomes tail call:

```
struct list {void *value;struct list *next;};
typedef struct list list;
void duplicate_aux(const list *ls, list *end);

list *duplicate(const list *ls)
{
    list head;

    duplicate_aux(ls, &head);
    return head.next;
}

void duplicate_aux(const list *ls, list *end)
{
    if (ls != NULL) {
        end->next = malloc(sizeof *end);
        end->next->value = ls->value;
        duplicate_aux(ls->next, end->next);
    } else {
        end->next = NULL;
    }
}
```

```
;; in Scheme,
(define (duplicate ls)
  (let ((head (list 1)))
    (let dup ((ls  ls)
              (end head))
      (cond
        ((not (null? ls))
         (set-cdr! end (list (car ls)))
         (dup (cdr ls) (cdr end)))))
    (cdr head)))
```

```
%% in Prolog,
dup([X|Xs],R):-
   R=[X|Ys],
   dup(Xs,Ys).
dup([],[]).
```

(A sentinel head node is used to simplify the code.) The callee now appends to the end of the growing list, rather than have the caller prepend to the beginning of the returned list. The work is now done on the way *forward* from the list's start, *before* the recursive call which then proceeds further, instead of *backward* from the list's end, *after* the recursive call has returned its result. It is thus similar to the accumulating parameter technique, turning a recursive computation into an iterative one.

Characteristically for this technique, a parent frame is created on the execution call stack, which the tail-recursive callee can reuse as its own call frame if the tail-call optimization is present.

The tail-recursive implementation can now be converted into an explicitly iterative form, as an accumulating loop:

```
struct list {void *value;struct list *next;};
typedef struct list list;

list *duplicate(const list *ls)
{
    list head, *end;
```

```
;; in Scheme,
(define (duplicate ls)
  (let ((head (list 1)))
    (do ((end head (cdr end))
         (ls  ls   (cdr ls )))
        ((null? ls) (cdr head))
```

```
    end = &head;
    while (ls != NULL)
    {
        end->next        = malloc(sizeof *end);
        end->next->value = ls->value;
        ls = ls->next;
        end = end->next;
    }
    end->next = NULL;
    return head.next;
}
```

```
        (set-cdr! end (list (car ls))))))
```

```
%% in Prolog,
%% N/A
```

History

In a paper delivered to the ACM conference in Seattle in 1977, Guy L. Steele summarized the debate over the GOTO and structured programming, and observed that procedure calls in the tail position of a procedure can be best treated as a direct transfer of control to the called procedure, typically eliminating unnecessary stack manipulation operations.[2] Since such "tail calls" are very common in Lisp, a language where procedure calls are ubiquitous, this form of optimization considerably reduces the cost of a procedure call compared to other implementations. Steele argued that poorly implemented procedure calls had led to an artificial perception that the GOTO was cheap compared to the procedure call. Steele further argued that "in general procedure calls may be usefully thought of as GOTO statements which also pass parameters, and can be uniformly coded as [machine code] JUMP instructions", with the machine code stack manipulation instructions "considered an optimization (rather than vice versa!)".[2] Steele cited evidence that well optimized numerical algorithms in Lisp could execute faster than code produced by then-available commercial Fortran compilers because the cost of a procedure call in Lisp was much lower. In Scheme, a Lisp dialect developed by Steele with Gerald Jay Sussman, tail call elimination is guaranteed to be implemented in any interpreter.[11]

Implementation methods

Tail recursion is important to some high-level languages, especially functional and logic languages and members of the Lisp family. In these languages, tail recursion is the most commonly used way (and sometimes the only way available) of implementing iteration. The language specification of Scheme requires that tail calls are to be optimized so as not to grow the stack. Tail calls can be made explicitly in Perl, with a variant of the "goto" statement that takes a function name: `goto &NAME;`[12]

However, for language implementations which store function arguments and local variables on a call stack (which is the default implementation for many languages, at least on systems with a hardware stack, such as the x86), implementing generalized tail call optimization presents an issue: if the size of the callee's activation record is different from that of the caller, then additional cleanup or resizing of the stack frame may be required. For these cases, optimizing tail recursion remains trivial, but general tail call optimization may be harder to implement efficiently.

For example, in the Java virtual machine (JVM), tail-recursive calls can be eliminated (as this reuses the existing call stack), but general tail calls cannot be (as this changes the call stack).[13][14] As a result, functional languages such as Scala that target the JVM can efficiently implement direct tail recursion, but not mutual tail recursion.

The GCC, LLVM/Clang, and Intel compiler suites perform tail call optimization for C and other languages at higher optimization levels or when the `-foptimize-sibling-calls` option is passed.[15][16][17] Though the given language syntax may not explicitly support it, the compiler can make this optimization whenever it can determine that the return types for the caller and callee are equivalent, and that the argument types passed to both function are either the same, or require the same amount of total storage space on the call stack.[18]

Various implementation methods are available.

In assembly

Tail calls are often optimized by interpreters and compilers of functional programming and logic programming languages to more efficient forms of iteration. For example, Scheme programmers commonly express while loops as calls to procedures in tail position and rely on the Scheme compiler or interpreter to substitute the tail calls with more efficient jump instructions.[19]

For compilers generating assembly directly, tail call elimination is easy: it suffices to replace a call opcode with a jump one, after fixing parameters on the stack. From a compiler's perspective, the first example above is initially translated into pseudo-assembly language (in fact, this is valid x86 assembly):

```
foo:
 call B
 call A
 ret
```

Tail call elimination replaces the last two lines with a single jump instruction:

```
foo:
 call B
 jmp  A
```

After subroutine `A` completes, it will then return directly to the return address of `foo`, omitting the unnecessary `ret` statement.

Typically, the subroutines being called need to be supplied with parameters. The generated code thus needs to make sure that the call frame for A is properly set up before jumping to the tail-called subroutine. For instance, on platforms where the call stack does not just contain the return address, but also the parameters for the subroutine, the compiler may need to emit instructions to adjust the call stack. On such a platform, for the code:

```
function foo(data1, data2)
```

```
    B(data1)
    return A(data2)
```

(where `data1` and `data2` are parameters) a compiler might translate that as:[b]

```
 1  foo:
 2    mov  reg,[sp+data1] ; fetch data1 from stack (sp) parameter into a scratch register.
 3    push reg            ; put data1 on stack where B expects it
 4    call B              ; B uses data1
 5    pop                 ; remove data1 from stack
 6    mov  reg,[sp+data2] ; fetch data2 from stack (sp) parameter into a scratch register.
 7    push reg            ; put data2 on stack where A expects it
 8    call A              ; A uses data2
 9    pop                 ; remove data2 from stack.
10    ret
```

A tail call optimizer could then change the code to:

```
1  foo:
2    mov  reg,[sp+data1] ; fetch data1 from stack (sp) parameter into a scratch register.
3    push reg            ; put data1 on stack where B expects it
4    call B              ; B uses data1
5    pop                 ; remove data1 from stack
6    mov  reg,[sp+data2] ; fetch data2 from stack (sp) parameter into a scratch register.
7    mov  [sp+data1],reg ; put data2 where A expects it
8    jmp  A              ; A uses data2 and returns immediately to caller.
```

This code is more efficient both in terms of execution speed and use of stack space.

Through trampolining

Since many Scheme compilers use C as an intermediate target code, the tail recursion must be encoded in C without growing the stack, even if the C compiler does not optimize tail calls. Many implementations achieve this by using a device known as a trampoline, a piece of code that repeatedly calls functions. All functions are entered via the trampoline. When a function has to tail-call another, instead of calling it directly and then returning the result, it returns the address of the function to be called and the call parameters back to the trampoline (from which it was called itself), and the trampoline takes care of calling this function next with the specified parameters. This ensures that the C stack does not grow and iteration can continue indefinitely.

It is possible to implement trampolines using higher-order functions in languages that support them, such as Groovy, Visual Basic .NET and C#.[20]

Using a trampoline for all function calls is rather more expensive than the normal C function call, so at least one Scheme compiler, Chicken, uses a technique first described by Henry Baker from an unpublished suggestion by Andrew Appel,[21] in which normal C calls are used but the stack size is checked before every call. When the stack reaches its maximum permitted size, objects on the stack are garbage-collected using the Cheney algorithm by moving all live data into a separate heap. Following this, the stack is unwound ("popped") and the program resumes from the state saved just before the garbage collection. Baker says "Appel's method avoids making a large number of small trampoline bounces by

occasionally jumping off the Empire State Building."[21] The garbage collection ensures that mutual tail recursion can continue indefinitely. However, this approach requires that no C function call ever returns, since there is no guarantee that its caller's stack frame still exists; therefore, it involves a much more dramatic internal rewriting of the program code: continuation-passing style.

Relation to *while* construct

Tail recursion can be related to the *while* control flow operator by means of a transformation such as the following:

```
function foo(x) is:
  if predicate(x) then
    return foo(bar(x))
  else
    return baz(x)
```

The above construct transforms to:

```
function foo(x) is:
  while predicate(x) do:
    x ← bar(x)
  return baz(x)
```

In the preceding, x may be a tuple involving more than one variable: if so, care must be taken in designing the assignment statement $x \leftarrow \text{bar}(x)$ so that dependencies are respected. One may need to introduce auxiliary variables or use a *swap* construct.

More general uses of tail recursion may be related to control flow operators such as **break** and **continue**, as in the following:

```
function foo(x) is:
  if p(x) then
    return bar(x)
  else if q(x) then
    return baz(x)
  ...
  else if t(x) then
    return foo(quux(x))
  ...
  else
    return foo(quuux(x))
```

where *bar* and *baz* are direct return calls, whereas *quux* and *quuux* involve a recursive tail call to *foo*. A translation is given as follows:

```
function foo(x) is:
  do:
    if p(x) then
      x ← bar(x)
      break
    else if q(x) then
      x ← baz(x)
      break
```

```
    ...
    else if t(x) then
      x ← quux(x)
      continue
    ...
    else
      x ← quuux(x)
      continue
    loop
  return x
```

By language

- Haskell - Yes[22]
- Common Lisp - Some implementations perform tail-call optimization during compilation if optimizing for speed
- JavaScript - ECMAScript 6.0 compliant engines should have tail calls[23] which is now implemented on Safari/WebKit[24]
- Lua - Tail recursion is performed by the reference implementation[25]
- Python - Stock Python implementations do not perform tail-call optimization, though a third-party module is available to do this.[26] Language inventor Guido van Rossum contends that stack traces are altered by tail call elimination making debugging harder, and prefers that programmers use explicit iteration instead[27]
- Rust - tail-call optimization may be done in limited circumstances, but is not guaranteed[28]
- Scheme - Required by the language definition[29][30]
- Racket - Yes[31]
- Tcl - Since Tcl 8.6, Tcl has a tailcall command[32]
- Kotlin - Has `tailrec` modifier for functions[33]
- Elixir - Elixir implements tail-call optimization[34] As do all languages currently targeting the BEAM VM.
- Perl - Explicit with a variant of the "goto" statement that takes a function name: `goto &NAME;`[35]
- Scala - Tail recursive functions are automatically optimized by the compiler. Such functions can also optionally be marked with a `@tailrec` annotation, which makes it a compilation error if the function is not tail recursive[36]
- Objective-C - Compiler optimizes tail calls when -O1 (or higher) option specified but it is easily disturbed by calls added by Automatic Reference Counting (ARC).
- F#- F# implements TCO by default where possible [37]
- Clojure - Clojure has `recur` special form.[38]

See also

- Course-of-values recursion
- Recursion (computer science)
- Inline expansion
- Leaf subroutine
- Corecursion

Notes

a. Like this:

```
if (ls != NULL) {
  head = malloc( sizeof *head);
  head->value = ls->value;
  head->next = duplicate( ls->next);
}
```

b. The `call` instruction first pushes the current code location onto the stack and then performs an unconditional jump to the code location indicated by the label. The `ret` instruction first pops a code location off the stack, then performs an unconditional jump to the retrieved code location.

References

1. Steven Muchnick; Muchnick and Associates (15 August 1997). *Advanced Compiler Design Implementation* (https://books.google.com/books?id=Pq7pHwG1_OkC&printsec=frontcover#v=onepage&q=%22tail%20call%22&f=false). Morgan Kaufmann. ISBN 978-1-55860-320-2.
2. Steele, Guy Lewis (1977). "Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, LAMBDA: The Ultimate GOTO". *Proceedings of the 1977 annual conference on - ACM '77*. pp. 153–162. doi:10.1145/800179.810196 (https://doi.org/10.1145%2F800179.810196). hdl:1721.1/5753 (https://hdl.handle.net/1721.1%2F5753). ISBN 978-1-4503-2308-6.
3. "The LLVM Target-Independent Code Generator — LLVM 7 documentation" (http://llvm.org/docs/CodeGenerator.html#sibling-call-optimization). *llvm.org*.
4. "recursion - Stack memory usage for tail calls - Theoretical Computer Science" (https://cstheory.stackexchange.com/q/7540). Cstheory.stackexchange.com. 2011-07-29. Retrieved 2013-03-21.
5. "Revised^6 Report on the Algorithmic Language Scheme" (http://www.r6rs.org/final/html/r6rs/r6rs-Z-H-8.html#node_sec_5.11). R6rs.org. Retrieved 2013-03-21.
6. "Revised^6 Report on the Algorithmic Language Scheme - Rationale" (http://www.r6rs.org/final/html/r6rs-rationale/r6rs-rationale-Z-H-7.html#node_sec_5.3). R6rs.org. Retrieved 2013-03-21.
7. "Revised^6 Report on the Algorithmic Language Scheme" (http://www.r6rs.org/final/html/r6rs/r6rs-Z-H-14.html#node_sec_11.20). R6rs.org. Retrieved 2013-03-21.
8. Sussman, G. J.; Abelson, Hal (1984). *Structure and Interpretation of Computer Programs* (https://archive.org/details/structureinterpr00abel). Cambridge, MA: MIT Press. ISBN 0-262-01077-1.
9. D. H. D. Warren, *DAI Research Report 141*, University of Edinburgh, 1980.
10. Daniel P. Friedman and David S. Wise, *Technical Report TR19: Unwinding Structured Recursions into Iterations* (http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR19), Indiana University, Dec. 1974.

11. R5RS Sec. 3.5, Richard Kelsey; William Clinger; Jonathan Rees; et al. (August 1998). "Revised^5 Report on the Algorithmic Language Scheme" (http://www.schemers.org/Documents/Standards/R5RS/). *Higher-Order and Symbolic Computation*. **11** (1): 7–105. doi:10.1023/A:1010051815785 (https://doi.org/10.1023%2FA%3A1010051815785).
12. Contact details. "goto" (http://perldoc.perl.org/functions/goto.html). perldoc.perl.org. Retrieved 2013-03-21.
13. "What is difference between tail calls and tail recursion? (https://stackoverflow.com/questions/12045299/what-is-difference-between-tail-calls-and-tail-recursion)", *Stack Overflow*
14. "What limitations does the JVM impose on tail-call optimization (http://programmers.stackexchange.com/questions/157684/what-limitations-does-the-jvm-impose-on-tail-call-optimization)", *Programmers Stack Exchange*
15. Lattner, Chris. "LLVM Language Reference Manual, section: The LLVM Target-Independent Code Generator, sub: Tail Call Optimization" (http://llvm.org/docs/CodeGenerator.html#tail-call-optimization). *The LLVM Compiler Infrastructure*. The LLVM Project. Retrieved 24 June 2018.
16. "Using the GNU Compiler Collection (GCC): Optimize Options" (https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html). *gcc.gnu.org*.
17. "foptimize-sibling-calls" (https://software.intel.com/en-us/node/522809). *software.intel.com*.
18. "Tackling C++ Tail Calls" (http://www.drdobbs.com/tackling-c-tail-calls/184401756).
19. Probst, Mark (20 July 2000). "proper tail recursion for gcc" (https://gcc.gnu.org/ml/gcc/2000-07/msg00595.html). GCC Project. Retrieved 10 March 2015.
20. Samuel Jack, Bouncing on your tail (http://blog.functionalfun.net/2008/04/bouncing-on-your-tail.html). *Functional Fun*. April 9, 2008.
21. Henry Baker, "CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A." (http://home.pipeline.com/~hbaker1/CheneyMTA.html)
22. "Tail recursion - HaskellWiki" (https://wiki.haskell.org/Tail_recursion). *wiki.haskell.org*. Retrieved 2019-06-08.
23. Beres-Deak, Adam. "Worth watching: Douglas Crockford speaking about the new good parts of JavaScript in 2014" (http://bdadam.com/blog/video-douglas-crockford-about-the-new-good-parts.html). *bdadam.com*.
24. "ECMAScript 6 in WebKit" (https://www.webkit.org/blog/4054/es6-in-webkit/). 13 October 2015.
25. "Lua 5.2 Reference Manual" (http://www.lua.org/manual/5.2/manual.html#3.4.9). *www.lua.org*.
26. "baruchel/tco" (https://github.com/baruchel/tco). *GitHub*.
27. Rossum, Guido Van (22 April 2009). "Neopythonic: Tail Recursion Elimination" (http://neopythonic.blogspot.com/2009/04/tail-recursion-elimination.html).
28. "Rust FAQ" (https://prev.rust-lang.org/en-US/faq.html#does-rust-do-tail-call-optimization). *prev.rust-lang.org*.
29. "Revised^5 Report on the Algorithmic Language Scheme" (http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-6.html#%25_sec_3.5). *www.schemers.org*.
30. "Revised^6 Report on the Algorithmic Language Scheme" (http://www.r6rs.org/final/html/r6rs/r6rs-Z-H-8.html#node_sec_5.11). *www.r6rs.org*.
31. "The Racket Reference" (https://docs.racket-lang.org/reference/eval-model.html#(part._.Tail_.Position)). *docs.racket-lang.org*.

32. "tailcall manual page - Tcl Built-In Commands" (http://www.tcl.tk/man/tcl/TclCmd/tailcall.htm). *www.tcl.tk*.
33. "Functions: infix, vararg, tailrec - Kotlin Programming Language" (https://kotlinlang.org/docs/reference/functions.html#tail-recursive-functions). *Kotlin*.
34. "Recursion" (http://elixir-lang.org/getting-started/recursion.html). *elixir-lang.github.com*.
35. "goto - perldoc.perl.org" (http://perldoc.perl.org/functions/goto.html). *perldoc.perl.org*.
36. "Scala Standard Library 2.13.0 - scala.annotation.tailrec" (https://www.scala-lang.org/api/2.13.0/scala/annotation/tailrec.html). *www.scala-lang.org*. Retrieved 2019-06-20.
37. "Tail Calls in F#" (https://blogs.msdn.microsoft.com/fsharpteam/2011/07/08/tail-calls-in-f/). *msdn*. Microsoft.
38. "(recur expr*)" (https://clojure.org/reference/special_forms#recur). *clojure.org*.
