-

-

-

Lazy evaluation

*Connected to:*

[Short-circuit evaluation](https://www.wikiwand.com/en/Short-circuit_evaluation)[Evaluation strategy](https://www.wikiwand.com/en/Evaluation_strategy)[Lookup table](https://www.wikiwand.com/en/Lookup_table)

|  |
| --- |
| [***Evaluation strategies***](https://www.wikiwand.com/en/Evaluation_strategy) |
| * [Eager evaluation](https://www.wikiwand.com/en/Eager_evaluation) * Lazy evaluation * [Partial evaluation](https://www.wikiwand.com/en/Partial_evaluation) * [Remote evaluation](https://www.wikiwand.com/en/Remote_evaluation) * [Short-circuit evaluation](https://www.wikiwand.com/en/Short-circuit_evaluation) |
| * [v](https://www.wikiwand.com/en/Template:Evaluation_strategy) * [t](https://www.wikiwand.com/en/Template_talk:Evaluation_strategy) * [e](https://en.wikipedia.org/w/index.php?title=Template:Evaluation_strategy&action=edit) |

In [programming language theory](https://www.wikiwand.com/en/Programming_language_theory), **lazy evaluation**, or **call-by-need**,[[1]](https://www.wikiwand.com/en/Lazy_evaluation#citenote1) is an [evaluation strategy](https://www.wikiwand.com/en/Evaluation_strategy) which delays the evaluation of an [expression](https://www.wikiwand.com/en/Expression_(computer_science)) until its value is needed ([non-strict evaluation](https://www.wikiwand.com/en/Non-strict_evaluation)) and which also avoids repeated evaluations ([sharing](https://www.wikiwand.com/en/Sharing_(computer_science))).[[2]](https://www.wikiwand.com/en/Lazy_evaluation#citenoteWattFindlay20042)[[3]](https://www.wikiwand.com/en/Lazy_evaluation#citenote3)

The benefits of lazy evaluation include:

* The ability to define [control flow](https://www.wikiwand.com/en/Control_flow) (structures) as abstractions instead of [primitives](https://www.wikiwand.com/en/Language_primitive).
* The ability to define [potentially infinite](https://www.wikiwand.com/en/Actual_infinity) [data structures](https://www.wikiwand.com/en/Data_structure). This allows for more straightforward implementation of some algorithms.
* The ability to define partially-defined data structures where some elements are errors. This allows for rapid prototyping.

Lazy evaluation is often combined with [memoization](https://www.wikiwand.com/en/Memoization), as described in [Jon Bentley](https://www.wikiwand.com/en/Jon_Bentley_(computer_scientist))'s *Writing Efficient Programs*.[[4]](https://www.wikiwand.com/en/Lazy_evaluation#citenote4) After a function's value is computed for that parameter or set of parameters, the result is stored in a [lookup table](https://www.wikiwand.com/en/Lookup_table) that is indexed by the values of those parameters; the next time the function is called, the table is consulted to determine whether the result for that combination of parameter values is already available. If so, the stored result is simply returned. If not, the function is evaluated and another entry is added to the lookup table for reuse.

Lazy evaluation is difficult to combine with imperative features such as [exception handling](https://www.wikiwand.com/en/Exception_handling) and [input/output](https://www.wikiwand.com/en/Input/output), because the order of operations becomes indeterminate.

The opposite of lazy evaluation is [eager evaluation](https://www.wikiwand.com/en/Eager_evaluation), sometimes known as strict evaluation. Eager evaluation is the evaluation strategy employed in most[[*quantify*](https://www.wikiwand.com/en/Wikipedia:Manual_of_Style/Dates_and_numbers)] [programming languages](https://www.wikiwand.com/en/Programming_language).

History

Lazy evaluation was introduced for [lambda calculus](https://www.wikiwand.com/en/Lambda_calculus) by Christopher Wadsworth[[5]](https://www.wikiwand.com/en/Lazy_evaluation#citenote5) and employed by the [Plessey System 250](https://www.wikiwand.com/en/Plessey_System_250) as a critical part of a Lambda-Calculus Meta-Machine, reducing the resolution overhead for access to objects in a capability-limited address space.[[6]](https://www.wikiwand.com/en/Lazy_evaluation#citenote6) For programming languages, it was independently introduced by Peter Henderson and [James H. Morris](https://www.wikiwand.com/en/James_H._Morris)[[7]](https://www.wikiwand.com/en/Lazy_evaluation#citenote7) and by [Daniel P. Friedman](https://www.wikiwand.com/en/Daniel_P._Friedman) and David S. Wise.[[8]](https://www.wikiwand.com/en/Lazy_evaluation#citenote8)[[9]](https://www.wikiwand.com/en/Lazy_evaluation#citenote9)

Applications

Delayed evaluation is used particularly in [**functional programming**](https://www.wikiwand.com/en/Functional_programming) languages. When using delayed evaluation, an expression is not evaluated as soon as it gets bound to a variable, but when the evaluator is forced to produce the expression's value. That is, a statement such as x = expression; (i.e. the assignment of the result of an expression to a variable) clearly calls for the expression to be evaluated and the result placed in x, but what actually is in x is irrelevant until there is a need for its value via a reference to x in some later expression whose evaluation could itself be deferred, though eventually the rapidly growing tree of dependencies would be pruned to produce some symbol rather than another for the outside world to see.[[10]](https://www.wikiwand.com/en/Lazy_evaluation#citenoteWadler200610)

Control structures

This section needs additional citations for [verification](https://www.wikiwand.com/en/Wikipedia:Verifiability). Please help [improve this article](https://en.wikipedia.org/w/index.php?title=Lazy_evaluation&action=edit) by [adding citations to reliable sources](https://www.wikiwand.com/en/Help:Referencing_for_beginners). Unsourced material may be challenged and removed. (March 2011) ([Learn how and when to remove this template message](https://www.wikiwand.com/en/Help:Maintenance_template_removal))

Lazy evaluation allows control structures to be defined normally, and not as primitives or compile-time techniques. For example one can define [if-then-else](https://www.wikiwand.com/en/If-then-else) and [short-circuit evaluation](https://www.wikiwand.com/en/Short-circuit_evaluation) operators:[[11]](https://www.wikiwand.com/en/Lazy_evaluation#citenote11)[[12]](https://www.wikiwand.com/en/Lazy_evaluation#citenote12)

ifThenElse **True** b c = b

ifThenElse **False** b c = c

-- **or**

**True** || b = **True**

**False** || b = b

-- **and**

**True** && b = b

**False** && b = **False**

These have the usual semantics, i.e. ifThenElse a b c evaluates (a), then if and only if (a) evaluates to true does it evaluate (b), otherwise it evaluates (c). That is, exactly one of (b) or (c) will be evaluated. Similarly for EasilyComputed || LotsOfWork, if the easy part gives **True** the lots of work expression could be avoided. Finally, when evaluating SafeToTry && Expression, if *SafeToTry* is **false** there will be no attempt at evaluating the *Expression*.

Conversely, in an eager language the above definition for ifThenElse a b c would evaluate (a), (b), and (c) regardless of the value of (a). This is not the desired behavior, as (b) or (c) may have [side effects](https://www.wikiwand.com/en/Side_effect_(computer_science)), take a long time to compute, or throw errors. It is usually possible to introduce user-defined lazy control structures in eager languages as functions, though they may depart from the language's syntax for eager evaluation: Often the involved code bodies need to be wrapped in a function value, so that they are executed only when called.

Working with infinite data structures

Delayed evaluation has the advantage of being able to create calculable infinite lists without infinite loops or size matters interfering in computation. The actual values are only computed when needed. For example, one could create a function that creates an infinite list (often called a [*stream*](https://www.wikiwand.com/en/Stream_(computing))) of [Fibonacci numbers](https://www.wikiwand.com/en/Fibonacci_number). The calculation of the *n*-th Fibonacci number would be merely the extraction of that element from the infinite list, forcing the evaluation of only the first n members of the list.[[13]](https://www.wikiwand.com/en/Lazy_evaluation#citenoteMtayer200213)[[14]](https://www.wikiwand.com/en/Lazy_evaluation#citenoteMachineryLanguages200214)

Take for example this trivial program in Haskell:

numberFromInfiniteList :: Int -> Int

numberFromInfiniteList n = infinity !! n - 1

where infinity = [1..]

main = print $ numberFromInfiniteList 4

In the function numberFromInfiniteList, the value of infinity is an infinite range, but until an actual value (or more specifically, a specific value at a certain index) is needed, the list is not evaluated, and even then it is only evaluated as needed (that is, until the desired index.) Provided the programmer is careful, the program completes normally. However, certain calculations may result in the program attempting to evaluate an infinite number of elements; for example, requesting the length of the list or trying to sum the elements of the list with a [fold operation](https://www.wikiwand.com/en/Fold_(higher-order_function)) would result in the program either failing to terminate or running [out of memory](https://www.wikiwand.com/en/Out_of_memory).

As another example, the list of all Fibonacci numbers can be written in the [Haskell](https://www.wikiwand.com/en/Haskell_(programming_language)) programming language as:[[14]](https://www.wikiwand.com/en/Lazy_evaluation#citenoteMachineryLanguages200214)

**fibs** = 0 : 1 : zipWith (+) fibs (tail fibs)

In Haskell syntax, ":" prepends an element to a list, tail returns a list without its first element, and zipWith uses a specified function (in this case addition) to combine corresponding elements of two lists to produce a third.[[13]](https://www.wikiwand.com/en/Lazy_evaluation#citenoteMtayer200213)

List-of-successes pattern

This section needs expansion. You can help by [adding to it](https://en.wikipedia.org/w/index.php?title=Lazy_evaluation&action=edit&section=). (March 2011)

Other uses

In computer [windowing systems](https://www.wikiwand.com/en/Windowing_system), the painting of information to the screen is driven by *expose events* which drive the display code at the last possible moment. By doing this, windowing systems avoid computing unnecessary display content updates.[[15]](https://www.wikiwand.com/en/Lazy_evaluation#citenoteLampson15)

Another example of laziness in modern computer systems is [copy-on-write](https://www.wikiwand.com/en/Copy-on-write) page allocation or [demand paging](https://www.wikiwand.com/en/Demand_paging), where memory is allocated only when a value stored in that memory is changed.[[15]](https://www.wikiwand.com/en/Lazy_evaluation#citenoteLampson15)

Laziness can be useful for high performance scenarios. An example is the Unix [mmap](https://www.wikiwand.com/en/Mmap) function, which provides *demand driven* loading of pages from disk, so that only those pages actually touched are loaded into memory, and unneeded memory is not allocated.

[MATLAB](https://www.wikiwand.com/en/MATLAB) implements [*copy on edit*](https://www.wikiwand.com/en/Copy_on_write), where arrays which are copied have their actual memory storage replicated only when their content is changed, possibly leading to an *out of memory* error when updating an element afterwards instead of during the copy operation.[[16]](https://www.wikiwand.com/en/Lazy_evaluation#citenote16)

Performance

The number of beta reductions to reduce a lambda term with call-by-need is no larger than the number needed by call-by-value or [call-by-name](https://www.wikiwand.com/en/Call-by-name) reduction.[[17]](https://www.wikiwand.com/en/Lazy_evaluation#citenote17)[[18]](https://www.wikiwand.com/en/Lazy_evaluation#citenote18) And with certain programs the number of steps may be much smaller, for example a specific family of lambda terms using [Church numerals](https://www.wikiwand.com/en/Church_numeral) take an infinite amount of steps with call-by-value (i.e. never complete), an exponential number of steps with call-by-name, but only a polynomial number with call-by-need. Call-by-need embodies two optimizations - never repeat work (similar to call-by-value), and never perform unnecessary work (similar to call-by-name).[[19]](https://www.wikiwand.com/en/Lazy_evaluation#citenoteStelle19) Lazy evaluation can also lead to reduction in memory footprint, since values are created when needed.[[20]](https://www.wikiwand.com/en/Lazy_evaluation#citenoteSmith200920)

In practice, lazy evaluation may cause significant performance issues compared to eager evaluation. For example, on modern computer architectures, delaying a computation and performing it later is slower than performing it immediately. This can be alleviated through [strictness analysis](https://www.wikiwand.com/en/Strictness_analysis).[[19]](https://www.wikiwand.com/en/Lazy_evaluation#citenoteStelle19) Lazy evaluation can also introduce [memory leaks](https://www.wikiwand.com/en/Memory_leak) due to unevaluated expressions.[[21]](https://www.wikiwand.com/en/Lazy_evaluation#citenoteFOOTNOTELaunchbury199321)[[22]](https://www.wikiwand.com/en/Lazy_evaluation#citenote22)

Implementation

Some programming languages delay evaluation of expressions by default, and some others provide [functions](https://www.wikiwand.com/en/Subroutine) or special [syntax](https://www.wikiwand.com/en/Syntax_of_programming_languages) to delay evaluation. In [Miranda](https://www.wikiwand.com/en/Miranda_(programming_language)) and [Haskell](https://www.wikiwand.com/en/Haskell_(programming_language)), evaluation of function arguments is delayed by default. In many other languages, evaluation can be delayed by explicitly suspending the computation using special syntax (as with [Scheme's](https://www.wikiwand.com/en/Scheme_(programming_language)) "delay" and "force" and [OCaml](https://www.wikiwand.com/en/OCaml)'s "lazy" and "Lazy.force") or, more generally, by wrapping the expression in a [thunk](https://www.wikiwand.com/en/Thunk_(delayed_computation)). The object representing such an explicitly delayed evaluation is called a [*lazy future*](https://www.wikiwand.com/en/Lazy_future)*.* [Raku](https://www.wikiwand.com/en/Raku_(programming_language)) uses lazy evaluation of lists, so one can assign infinite lists to variables and use them as arguments to functions, but unlike Haskell and Miranda, Raku does not use lazy evaluation of arithmetic operators and functions by default.[[10]](https://www.wikiwand.com/en/Lazy_evaluation#citenoteWadler200610)

Laziness and eagerness

Controlling eagerness in lazy languages

In lazy programming languages such as Haskell, although the default is to evaluate expressions only when they are demanded, it is possible in some cases to make code more eager—or conversely, to make it more lazy again after it has been made more eager.

This can be done by explicitly coding something which forces evaluation (which may make the code more eager) or avoiding such code (which may make the code more lazy). *Strict* evaluation usually implies eagerness, but they are technically different concepts.

However, there is an optimisation implemented in some compilers called [strictness analysis](https://www.wikiwand.com/en/Strictness_analysis), which, in some cases, allows the compiler to infer that a value will always be used. In such cases, this may render the programmer's choice of whether to force that particular value or not, irrelevant, because strictness analysis will force strict evaluation.

In Haskell, marking constructor fields strict means that their values will always be demanded immediately. The seq function can also be used to demand a value immediately and then pass it on, which is useful if a constructor field should generally be lazy. However, neither of these techniques implements *recursive* strictness—for that, a function called deepSeq was invented.

Also, [pattern matching](https://www.wikiwand.com/en/Pattern_matching) in Haskell 98 is strict by default, so the [~](https://www.wikiwand.com/en/Tilde#Computer_languages) qualifier has to be used to make it lazy.[[23]](https://www.wikiwand.com/en/Lazy_evaluation#citenote23)

Simulating laziness in eager languages

Java

In [Java](https://www.wikiwand.com/en/Java_(programming_language)), lazy evaluation can be done by using objects that have a method to evaluate them when the value is needed. The body of this method must contain the code required to perform this evaluation. Since the introduction of [lambda expressions](https://www.wikiwand.com/en/Anonymous_function) in Java SE8, Java has supported a compact notation for this. The following example [generic](https://www.wikiwand.com/en/Generic_classes_in_Java) interface provides a framework for lazy evaluation:[[24]](https://www.wikiwand.com/en/Lazy_evaluation#citenotePiwowarek201824)[[25]](https://www.wikiwand.com/en/Lazy_evaluation#citenoteJones202025)

interface Lazy<T> {

T eval();

}

The Lazy interface with its eval() method is equivalent to the Supplier interface with its get() method in the java.util.function library.[[26]](https://www.wikiwand.com/en/Lazy_evaluation#citenote26)

Each class that implements the Lazy interface must provide an eval method, and instances of the class may carry whatever values the method needs to accomplish lazy evaluation. For example, consider the following code to lazily compute and print 210:

Lazy<Integer> **a** = **()** -> 1;

**for** (int i = 0; i < 10; i++) {

Lazy<Integer> b = a;

**a** = **()** -> b.eval() + b.eval();

}

System.out.println("a = " + a.eval());

In the above, the variable a initially refers to a lazy integer object created by the lambda expression () -> 1. Evaluating this lambda expression is similar[[a]](https://www.wikiwand.com/en/Lazy_evaluation#citenoteJavaLambda27) to constructing a new instance of an [anonymous class](https://www.wikiwand.com/en/Anonymous_class) that implements Lazy<Integer> with an eval method returning 1.

Each iteration of the loop links a to a new object created by evaluating the lambda expression inside the loop. Each of these objects holds a reference to another lazy object, b, and has an eval method that calls b.eval() twice and returns the sum. The variable b is needed here to meet Java's requirement that variables referenced from within a lambda expression be effectively final.

This is an inefficient program because this implementation of lazy integers does not [memoize](https://www.wikiwand.com/en/Memoize) the result of previous calls to eval. It also involves considerable [autoboxing and unboxing](https://www.wikiwand.com/en/Autoboxing). What may not be obvious is that, at the end of the loop, the program has constructed a [linked list](https://www.wikiwand.com/en/Linked_list) of 11 objects and that all of the actual additions involved in computing the result are done in response to the call to a.eval() on the final line of code. This call [recursively](https://www.wikiwand.com/en/Recursion_(computer_science)) traverses the list to perform the necessary additions.

We can build a Java class that memoizes a lazy objects as follows:[[24]](https://www.wikiwand.com/en/Lazy_evaluation#citenotePiwowarek201824)[[25]](https://www.wikiwand.com/en/Lazy_evaluation#citenoteJones202025)

**class** **Memo**<**T**> **implements** **Lazy**<**T**> {

**private** Lazy<T> lazy; *// a lazy expression, eval sets it to null*

**private** T memo; *// the memorandum of the previous value*

**public** Memo(Lazy<T> lazy) {

this.lazy = lazy;

}

**public** T **eval**() {

**if** (lazy != **null**) {

memo = lazy.**eval**();

lazy = **null**;

}

**return** memo;

}

}

This allows the previous example to be rewritten to be far more efficient. Where the original ran in time exponential in the number of iterations, the memoized version runs in linear time:

Lazy<Integer> **a** = **()** -> 1;

**for** (int i = 0; i < 10; i++) {

Lazy<Integer> b = a;

a = **new** Memo<Integer>**(() -> b.eval() + b.eval())**;

}

**System**.**out**.**println("a = " + a.eval())**;

Note that Java's lambda expressions are just [syntactic sugar](https://www.wikiwand.com/en/Syntactic_sugar). Anything you can write with a lambda expression can be rewritten as a call to construct an instance of an anonymous [inner class](https://www.wikiwand.com/en/Inner_class) implementing the interface,[[a]](https://www.wikiwand.com/en/Lazy_evaluation#citenoteJavaLambda27) and any use of an anonymous inner class can be rewritten using a named inner class, and any named inner class can be moved to the outermost nesting level.

JavaScript

In [JavaScript](https://www.wikiwand.com/en/JavaScript), lazy evaluation can be simulated by using a [generator](https://www.wikiwand.com/en/Generator_(computer_programming)). For example, the [stream](https://www.wikiwand.com/en/Stream_(computing)) of all [Fibonacci numbers](https://www.wikiwand.com/en/Fibonacci_numbers) can be written, using [memoization](https://www.wikiwand.com/en/Memoization), as:

*/\*\**

*\* Generator functions return generator objects, which reify lazy evaluation.*

*\* @return {!Generator<bigint>} A non-null generator of integers.*

*\*/*

function\* fibonacciNumbers() {

let memo = [1n, -1n]; *// create the initial state (e.g. a vector of "negafibonacci" numbers)*

**while** (**true**) { *// repeat indefinitely*

memo = [memo[0] + memo[1], memo[0]]; *// update the state on each evaluation*

**yield** memo[0]; *// yield the next value and suspend execution until resumed*

}

}

let stream = fibonacciNumbers(); *// create a lazy evaluated stream of numbers*

let first10 = Array.from(**new** Array(10), () => stream.next().value); *// evaluate only the first 10 numbers*

console.log(first10); *// the output is [0n, 1n, 1n, 2n, 3n, 5n, 8n, 13n, 21n, 34n]*

Python

In [Python](https://www.wikiwand.com/en/Python_(programming_language)) 2.x the range() function[[27]](https://www.wikiwand.com/en/Lazy_evaluation#citenote28) computes a list of integers. The entire list is stored in memory when the first assignment statement is evaluated, so this is an example of eager or immediate evaluation:

>>> r = range(10)

>>> print r

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> print r[3]

3

In Python 3.x the range() function[[28]](https://www.wikiwand.com/en/Lazy_evaluation#citenote29) returns a [generator](https://www.wikiwand.com/en/Generator_(computer_programming)) which computes elements of the list on demand. Elements are only generated when they are needed (e.g., when print(r[3]) is evaluated in the following example), so this is an example of lazy or deferred evaluation:

>>> r = range(10)

>>> print(r)

range(0, 10)

>>> print(r[3])

3

This change to lazy evaluation saves execution time for large ranges which may never be fully referenced and memory usage for large ranges where only one or a few elements are needed at any time.

In Python 2.x is possible to use a function called xrange() which returns an object that generates the numbers in the range on demand. The advantage of xrange is that generated object will always take the same amount of memory.

>>> r = xrange(10)

>>> print(r)

xrange(10)

>>> lst = [x for x in r]

>>> print(lst)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

From version 2.2 forward, Python manifests lazy evaluation by implementing iterators (lazy sequences) unlike tuple or list sequences. For instance (Python 2):

>>> numbers = range(10)

>>> iterator = iter(numbers)

>>> print numbers

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> print iterator

<listiterator object at 0xf7e8dd4c>

>>> print iterator.next()

0

The above example shows that lists are evaluated when called, but in case of iterator, the first element '0' is printed when need arises.

.NET Framework

**In the**[**.NET Framework**](https://www.wikiwand.com/en/.NET_Framework)**it is possible to do lazy evaluation using the class System.Lazy<T>.**[**[29]**](https://www.wikiwand.com/en/Lazy_evaluation#citenote30)

The class can be easily exploited in [F#](https://www.wikiwand.com/en/F_Sharp_(programming_language)) using the lazy keyword, while the force method will force the evaluation. There are also specialized collections like Microsoft.FSharp.Collections.Seq that provide built-in support for lazy evaluation.

let fibonacci = Seq.unfold **(fun (x, y) -> Some(x, (y, x + y)))** **(0I,1I)**

**fibonacci** |> **Seq**.**nth** 1000

In C# and VB.NET, the class System.Lazy<T> is directly used.

**public** **int** **Sum**()

{

**int** a = 0;

**int** b = 0;

Lazy<**int**> x = **new** Lazy<**int**>(() => a + b);

a = 3;

b = 5;

**return** x.Value; *// returns 8*

}

Or with a more practical example:

*// recursive calculation of the n'th fibonacci number*

**public** **int** **Fib**(**int** n)

{

**return** (n == 1)? 1 : (n == 2)? 1 : Fib(n-1) + Fib(n-2);

}

**public** **void** **Main**()

{

Console.WriteLine("Which Fibonacci number do you want to calculate?");

**int** n = Int32.Parse(Console.ReadLine());

Lazy<**int**> fib = **new** Lazy<**int**>(() => Fib(n)); *// function is prepared, but not executed*

**bool** execute;

**if** (n > 100)

{

Console.WriteLine("This can take some time. Do you really want to calculate this large number? [y/n]");

execute = (Console.ReadLine() == "y");

}

**else** execute = **true**;

**if** (execute) Console.WriteLine(fib.Value); *// number is only calculated if needed*

}

Another way is to use the **yield** keyword:

*// eager evaluation*

**public** IEnumerable<**int**> **Fibonacci**(**int** x)

{

IList<**int**> fibs = **new** List<**int**>();

**int** prev = -1;

**int** next = 1;

**for** (**int** i = 0; i < x; i++)

{

**int** sum = prev + next;

prev = next;

next = sum;

fibs.Add(sum);

}

**return** fibs;

}

*// lazy evaluation*

**public** IEnumerable<**int**> **LazyFibonacci**(**int** x)

{

**int** prev = -1;

**int** next = 1;

**for** (**int** i = 0; i < x; i++)

{

**int** sum = prev + next;

prev = next;

next = sum;

**yield** **return** sum;

}

}

-

-

-