Evaluation Strategies:

↳ Parameter-passing strategy.

```
func(x, y) {
}
```

```
Main() {
    a =
    b =
}
```

→ passed-into func

1. Binding Strategy
2. Evaluation Order
   - ↙ Strict {applicative order}
   - ↘ Non-Strict. {Normal order, Lazy eval}

Evaluation Strategy: set of rules for evaluating any expressions!

1. **Parameter Passing Strategy:**
   Defines the kind of value → passed into the function for each parameter.

2. **Binding Strategy:**
   The binding strategy figures out how to BIND the variables based on what type of variable is passed

3. **Evaluation Order:**
   Evaluation order decides how to process the arguments passed in and whether to even accept them at all! If accepted → then in what order?

Evaluation strategy

13 languages

Article Talk | Read Edit View history Tools

From Wikipedia, the free encyclopedia

In a programming language, an **evaluation strategy** is a set of rules for evaluating expressions.[1] The term is often used to refer to the more specific notion of a *parameter-passing strategy*[2] that defines the **kind of value** that is passed to the function for each parameter **(the *binding strategy*)**[3] and **whether to evaluate the parameters of a function call, and if so in what order (the *evaluation order*).**[4] ~~The notion of reduction strategy is distinct,[5] although some authors conflate the two terms and the definition of each term is not widely agreed upon.~~[6]

Evaluation strategies

- Lazy evaluation
- Partial evaluation
- Remote evaluation
- Short-circuit evaluation

V·T·E

> To illustrate, executing a function call `f(a,b)` may first evaluate the arguments `a` and `b`, store the results in references or memory locations `ref_a` and `ref_b`, then evaluate the function's body with those references passed in. This gives the function the ability to look up the original argument values passed in through dereferencing the parameters (some languages use specific operators to perform this), to modify them via assignment as if they were local variables, and to return values via the references. This is the call-by-reference evaluation strategy.[7]

Evaluation strategy is part of the semantics of the programming language definition. Some languages, such as PureScript, have variants with different evaluation strategies. Some declarative languages, such as Datalog, support multiple evaluation strategies. Some languages define a calling convention. [*clarification needed*]

→ "Binding Strategy"

Call by reference:

fnc(&a, &b)

arguments.

⇒ Calling the function:

1. Evaluates Arguments.

Remember!

x and y hold mem. add locations.

Fnc. Def:

fnc(*x, *y)

x = ref_a

y = ref_b

Dereference in fnc body to get value of a, b!

*x = a

*y = b

→ Must deref. mem add to get val of a/b.

fnc(int *x, int *y)

∴ *x = a

*y = b

because of x = &a, y = &b

→ *(&a) = a

→ *(&b) = b

Strict evaluation [edit]

Applicative order is a family of evaluation orders in which a function's arguments are evaluated completely before the function is applied.[22] This has the effect of making the function strict, i.e. the function's result is undefined if any of the arguments are undefined, so applicative order evaluation is more commonly called **strict evaluation**. Furthermore, a function call is performed as soon as it is encountered in a procedure, so it is also called **eager evaluation** or **greedy evaluation**.[23][24] Some authors refer to strict evaluation as "call by value" due to the call-by-value binding strategy requiring strict evaluation.[4]

Common Lisp, Eiffel and Java evaluate function arguments left-to-right. C leaves the order undefined.[25] Scheme requires the execution order to be the sequential execution of an unspecified permutation of the arguments.[26] OCaml similarly leaves the order unspecified, but in practice evaluates arguments right-to-left due to the design of its abstract machine.[27] All of these are strict evaluation.

→ Args eval completely before fnc. applied.

args undef? → res. undef.

fnc call performed asap. { EAGER eval } or "GREEDY Eval"

Non-strict evaluation [edit]

A **non-strict evaluation order** is an evaluation order that is not strict, that is, a function may return a result before all of its arguments are fully evaluated.[28]:46–47 The prototypical example is **normal order evaluation**, which does not evaluate any of the arguments until they are needed in the body of the function.[29]

> Normal order evaluation has the property that it **terminates without error** whenever **any other evaluation order would have terminated without error.**

[30] The name "normal order" comes from the lambda calculus, where normal order reduction will find a normal form if there is one (it is a "normalizing" reduction strategy).[31] Lazy evaluation is classified in this article as a binding technique rather than an evaluation order. But this distinction is not always followed and some authors define lazy evaluation as normal order evaluation or vice-versa,[22][32] or confuse non-strictness with lazy evaluation.[28]:43–44

fnc. returning result before args. eval.

NORMAL ORDER EVALUATION → does not eval. args. until needed. terminates. without error → even if error present.

BOOLEAN EXPRESSIONS! → functions as well.

a == 2 OR b == 3

always one true → ∴

All conditions are boolean expressions, but not all boolean expressions are conditions. There's not much more to this.

5 Award Share ...

Boolean expressions in many languages use a form of non-strict evaluation called short-circuit evaluation, where

> evaluation evaluates the left expression but may **skip the right expression if the result can be determined**—for example, in a disjunctive expression (OR) where `true` is encountered

, or in a conjunctive expression (AND) where `false` is encountered, and so forth.[32] **Conditional expressions similarly use non-strict evaluation - only one of the branches is evaluated**.[28]

Comparing → non-strict vs. strict evaluation:
{ applicative order }

Comparison of applicative order and normal order evaluation [edit]

> With normal order evaluation, expressions containing an expensive computation, an error, or an infinite loop will be ignored if not needed,[4] allowing the specification of user-defined control flow constructs, a facility not available with applicative order evaluation.

No exp. comp., error, infinite loop ignored.

Normal order evaluation uses complex structures such as thunks for unevaluated expressions, compared to the call stack used in applicative order evaluation.[33] Normal order evaluation has historically had a lack of usable debugging tools due to its complexity.[34]