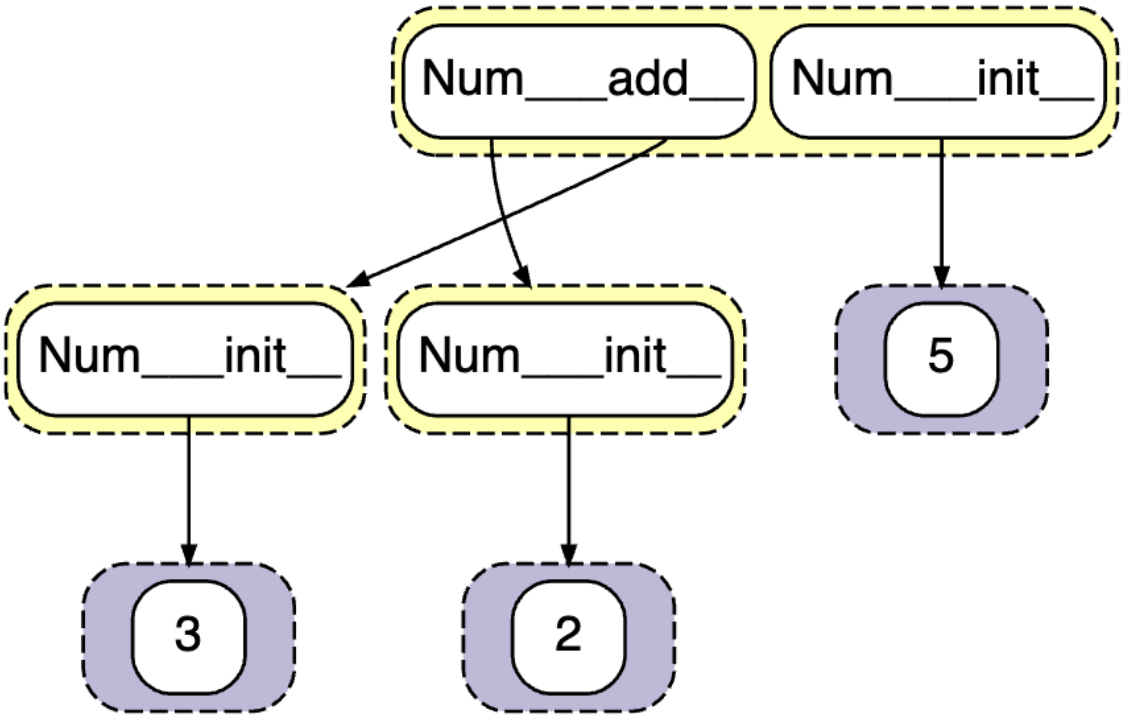


E-graph for Num(2) + Num(3) is equivalent to Num(5)

Step 1: The smallest possible egglog program

First off, I used Python 3.11 and egglog library version 12.0.0. Create a file (call it whatever you want, e.g. step01.py) and type this:

```
from egglog import * egraph = EGraph()
print(egraph.extract(i64(42)))
```

What each line means:

- `from egglog import *` imports everything from the egglog library
- `EGraph()` creates an empty e-graph, the core data structure that stores expressions and their equivalences
- `i64(42)` wraps the Python integer 42 into egglog's built-in 64-bit integer type. egglog has its own type system separate from Python's
- `egraph.extract(...)` pulls a value back out of the e-graph. "Extraction" means "give me the simplest equivalent expression"

By running `python step01.py` we get the output `42` . Makes sense—the simplest expression of 42 is 42.

Step 2: Defining a custom expression type

```
# We'll need this for self-referencing type annotations
from __future__ import annotations
from egglog import *   egraph = EGraph() class Num(Expr):
  def __init__(self, value: i64Like) -> None: … a = Num(42)
print(a)
print(egraph.extract(a))
```

New code added:

- `class Num(Expr)` defines a new sort (egglog's word for type). Inheriting from `Expr` means "this is a kind of symbolic expression called Num."
- `def __init__(self, value: i64Like) -> None: ...` declares that a Num can be created from an integer. The body is … (ellipsis)—this is not a real Python function. It's a declaration. It tells egglog "this operation exists" and egglog builds the implementation. Fun fact: … in python is the same as `pass`, so you could use either.
- `Num(42)` creates a symbolic expression representing the number 42.

Why not just use a Python int?

`Num(42)` doesn't compute anything. It creates a node in the e-graph. That's the whole point. A Python int is a value: 2 * 3 immediately becomes 6, and the original expression is gone. An e-graph node is a symbolic expression: `Num(2) * Num(3)` stays as "2 times 3" in the graph. Later, rewrite rules can prove that it equals `Num(6)`, but both representations are kept. The e-graph remembers every equivalent form, which is what makes it possible to search for the simplest or most efficient version of an expression.

The output we get from running the above code:

```
Num(42)
Num(42)
```

Both lines print `Num(42)`. The expression goes in, and extract pulls it back out unchanged—there's nothing simpler that it's equivalent to yet.

Step 3: Symbolic variables

```
from __future__ import annotations
from egglog import *   egraph = EGraph() class Num(Expr):
  def __init__(self, value: i64Like) -> None: …
  @classmethod
  def var(cls, name: StringLike) -> Num: … a = Num(42)
x = Num.var("x")
print(a)
print(x)
```

What's new:

- `@classmethod + def var(cls, name: StringLike) -> Num: ...` declares a second way to create a Num. Instead of a concrete number, this creates a named symbolic variable. Again the body is …—just a declaration, not an implementation.
- Num.var("x")—creates a symbolic variable called "x". It doesn't have a value. It just represents "some number called x."

A sort can have multiple constructors. Num(42) makes a concrete constant. Num.var("x") makes an unbound variable. Both are Num expressions, both live as nodes in the e-graph, but one holds a value and the other holds a name.

Running `python step03.py` gives:

```
Num(42)
Num.var("x")
```

The print output reflects exactly how each node was constructed—egglog remembers the structure, not a computed result.

Step 4: Naming expressions with egraph.let

```
from __future__ import annotations
from egglog import *  egraph = EGraph() class Num(Expr):
  def __init__(self, value: i64Like) -> None: …
  @classmethod
  def var(cls, name: StringLike) -> Num: …  a = egraph.let("a", Num(42))
x_w_label = egraph.let("x_label", Num.var("x"))  # Different names for x to see what is
printed
# - conventionally use the same name
print(a)
print(x_w_label)
print(egraph.extract(a))
print(egraph.extract(x_w_label))
```

What's new:

- `egraph.let("a", Num(42))`inserts the expression `Num(42)` into the e-graph and gives it the name "a". It returns a reference (a handle) to that expression.
- The difference between `Num(42)`alone and `egraph.let("a", Num(42))` is that let explicitly inserts the expression into the e-graph and labels it. The label is how the e-graph refers to it internally.

Running `python step04.py` gives:

```
a
x_label
Num(42)
Num.var("x")
Python
```

Printing the reference directly (print(a)) shows the label—just the name “a”. Printing `egraph.extract(a)` shows the actual expression stored in the e-graph: `Num(42)`. The label is a short handle. Extraction gives back the full symbolic expression.

Step 5: Adding operations

```
from __future__ import annotations
from egglog import *  egraph = EGraph() class Num(Expr):
  def __init__(self, value: i64Like) -> None: …
  @classmethod
  def var(cls, name: StringLike) -> Num: …
  def __add__(self, other: Num) -> Num: …
  def __mul__(self, other: Num) -> Num: … x = Num.var("x")
expr1 = egraph.let("expr1", Num(2) * (x * Num(3)))
expr2 = egraph.let("expr2", Num(6) * x)
print(egraph.extract(expr1))
print(egraph.extract(expr2))
```

In normal Python, def __add__(self, other): needs a body that says how to compute the result. Like return self.value + other.value. By overriding the `+` (`__add__`) operation we are letting egglog intercept the operation and add it to the e-graph. Num(2) + Num(3) doesn’t become 5. It becomes a node in the e-graph that says “there is an add, with a left child 2 and a right child 3.” Egglog’s functionality is not evaluating expressions but representing the different expressions of it, so the operations in expression is exactly what we are declaring here.

Code we added:

- `def __add__(self, other: Num) -> Num: ...` declares that two `Num` expressions can be combined with `+`. The result is another `Num`. Body is `...`
- def __mul__(self, other: Num) -> Num: …—same for *. Num(2) * (x * Num(3))—this does not compute 6x. It builds a tree of symbolic nodes: “2 times (x times 3)”. The * operator creates new e-graph nodes instead of doing arithmetic.

Running `python step05.py` gives:

```
Num(2) * (Num.var("x") * Num(3))
Num(6) * Num.var("x")
```

The two expressions extract exactly as they were put in. The e-graph has no reason to think they’re related because no rewrite rules exist yet.

Step 6: Rewrite rules—what egglog is for!

```
from __future__ import annotations
from egglog import * egraph = EGraph() class Num(Expr):
    def __init__(self, value: i64Like) -> None: pass     @classmethod
    def var(cls, name: StringLike) -> Num: pass     def __add__(self, other: Num) -> Num:
pass
    def __mul__(self, other: Num) -> Num: pass     a = egraph.let("a", Num(2) +
Num(3)) print("before:", egraph.extract(a)) p, q = vars_("p q", i64)
egraph.register(
        rewrite(Num(p) + Num(q)).to(Num(p + q)),
)
egraph.run(1) print("after:", egraph.extract(a))
```

The code added:

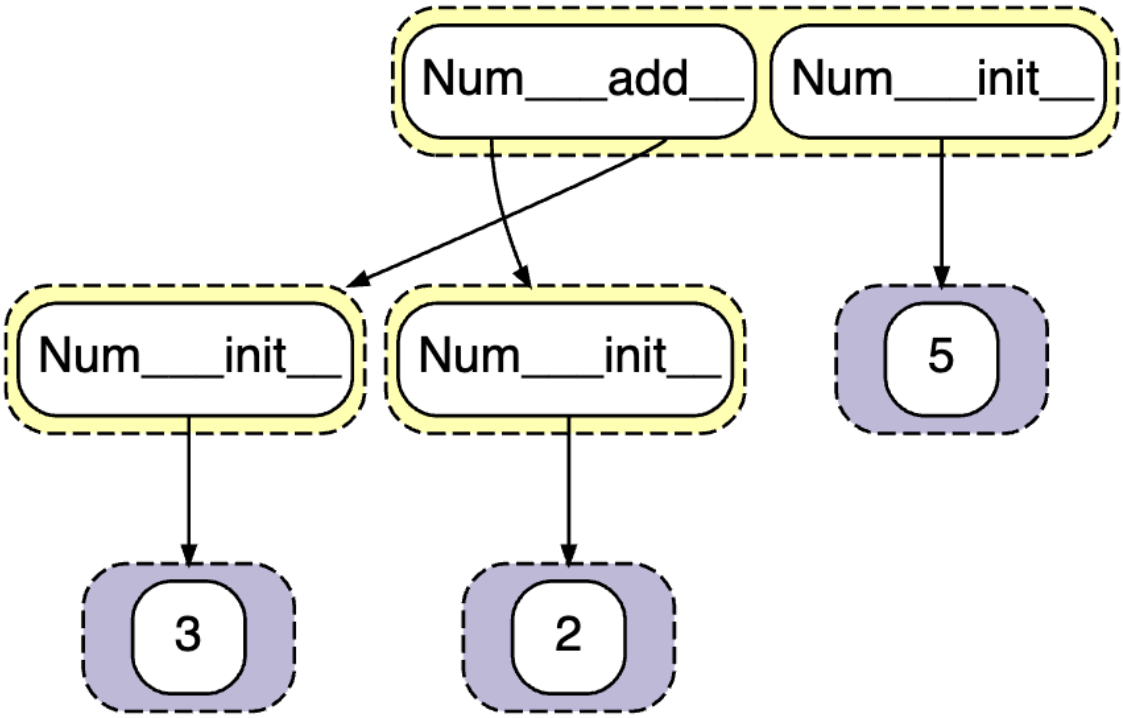
- `p, q = vars_("p q", i64)` creates pattern variables. These are wildcards that match any i64 value in the e-graph. The string `"p q"`is split by spaces to create two variables. From source code you can see how it is splitting by white space and creating RunTimeExpr variables that are assigned to `p` and `q`
- `rewrite(Num(p) + Num(q)).to(Num(p+q))` declares a rule: “whenever `Num(p) + Num(q)` appears in the e-graph where `p` and `q` are concrete integers, mark it as equivalent to `Num(p+q)`.” On the left side, `p` and `q`are wildcards matching any integer. On the right side, `p+q` is actual integer addition. Egglog computes the sum and wraps it back in `Num`. Under the hood, `rewrite()` stores the left-hand side, then `.to()` packages both sides into a `RewriteDecl`, a data structure holding two expression trees. This is just a Python object. Nothing has been sent to the engine yet.
- `egraph.register(...)` converts the `RewriteDecl` into a Rust-side command and sends it to the egglog engine. The rule is now stored in the engine, but has not been applied to any expressions yet.
- `egraph.run(1)` tells the Rust engine to scan every expression in the e-graph, pattern-match against every registered rule, and for each match add the right-hand side as an equivalent expression. The 1 means do one pass. When the engine finds `Num(2) + Num(3)` and matches it against the left-hand pattern, `p` binds to 2 and `q`binds to 3, then `Num(p+q)` becomes `Num(5)`, which gets merged into the same e-class as `Num(2) + Num(3)`.

Indeed, the output of the program is:

```
before: Num(2) + Num(3)
after: Num(5)
```

With `f.write(`egraph._graphviz().pipe(format=”svg”, encoding=”utf-8")`)` where `f` is a file descriptor we can visualize the mini e-graph. Note how `Num` initialized to 5 is in the same e-class as if just two `Num` objects were initialized to 2 and 3 and then added. Particularly, the `__add__`

operation is not commutative in this e-graph currently. That is, we do not have a rewrite rule for adding `Num(3) + Num(2)` to the same e-class yet.



E-graph for Num(2) + Num(3) is equivalent to Num(5)

That is the simplest working example for egglog library which manages defining nodes and allowed rewrites, as well as a simple way to group equivalent expressions into classes.