Metaprogramming for Software Correctness and Verification in Lean

Logic and Computation for Mathematicians Final Project

Salamun Nuhin

December 2025

Abstract

This project explores how using metaprogramming in `Lean 4` helps with software verification and correctness. The work for this project includes implementing verified data structures and algorithms, starting with stacks and queues and then moving onto insertion sort on lists. For each implementation, correctness properties are proven through formal proofs in `Lean 4` and then custom macros are written to automate repeated proof patterns. The macros include tactic macros that expand into sequences of proof steps and command macros that generate complete theorem declarations.

Contents

1 Introduction

One of the main themes from this course is the challenge of connecting mathematical proofs with formal verification. Even when an algorithm looks correct on paper, the actual implementation can have subtle bugs, unhandled edge cases or incorrect assumptions. This is where software verification as it tries to close this gap by expressing correctness properties as formal specifications and then mechanically checking that the code satisfies those specifications [5].

However, traditional verification work faces a practical problem. Writing proofs in a theorem prover is often very tedious and repetitive. When the same proof pattern shows up in many places, the standard approach is to copy and paste the same tactic sequences with small changes. This approach is both error prone and makes the proof code hard to read and maintain.

Metaprogramming provides a solution to this problem. The basic idea is to write code that generates other code automatically. In theorem proving, this means writing macros and tactics that generate proof scripts. `Lean 4` has a powerful metaprogramming system that lets users extend the language with custom syntax and automation [2, 6, 3].

1.1 Project goals

This project has two main goals. The first is to learn how to write verified code in `Lean 4` by building two case studies: basic data structures (stacks and queues) and a standard algorithm (insertion sort). The second goal is to understand how metaprogramming can make the verification easier by making custom macros that automate the repetitive parts of proofs.

The code is organized in two files.

- **`stacks&queues.lean`**: defines a stack and a queue with correctness lemmas and several metaprogramming examples including tactic macros and a command macro.
- **`InsertionSortProof.lean`**: implements insertion sort on lists of natural numbers with full correctness proofs (sortedness and permutation) and an alternative version of the proofs using custom macros.

1.2 What correctness means

In this context, correctness means that a piece of code satisfies a formal specification. For data structures like stacks and queues, the specification is usually a set of equations that describe how operations interact. An example would be if an element is pushed onto a stack and then immediately popped which would result in that element and the original stack.

For sorting algorithms on the other hand, correctness requires two properties:

- **Sortedness**: the output list is ordered according to some comparison relation.
- **Permutation**: the output list contains exactly the same elements as the input list.

Both properties are necessary to prove their correctness. Sortedness alone would allow an implementation that always returns an empty list. Permutation alone would allow an implementation that just returns the input unchanged. Together these two properties fully capture what sorting correctly means.

1.3 Why metaprogramming matters for software verification

At first, metaprogramming may seem like just a convenience feature that makes proofs shorter. However, it turns out to be much more important than that. There are three main reasons why metaprogramming is important for verification work.

First, metaprogramming reduces proof bugs. When the same proof pattern is written by hand multiple times, mistakes are likely. Steps might be forgotten or tactics might be applied in the wrong order. By putting the pattern into a macro, it only needs to be correct once. After that all uses of the macro will automatically be correct.

Second, metaprogramming creates abstraction boundaries. Proof scripts in theorem provers can become very long sequences of low level tactics. Macros allow these patterns to be given names and treated like higher level operations which makes proof scripts much easier to read and understand.

Third, metaprogramming makes verification scale. In actual software verification, there might be hundreds of similar definitions that all need similar lemmas. Writing all those proofs by hand is not realistic (which is true for dealing with abstract data types for lists). Command macros can generate families of theorems automatically, which is the only practical way to handle verification at a larger scale.

2 Background on Lean and metaprogramming

2.1 Lean as a programming language and proof assistant

`Lean 4` is both a functional programming language and an interactive theorem prover. The system is based on dependent type theory, which means that types can depend on values and propositions are represented as types. In this framework, a proof is just a term that has a proposition as its type. This design provides a unified environment where code and proofs exist in the same language and are checked by the same type checker.

This project uses several key features of `Lean 4`.

Induction and pattern matching. Most proofs about recursive data structures like lists are done by induction on the structure. `Lean 4` supports this through the `induction` tactic, which generates one subgoal for each constructor.

The `simp` tactic. The `simp` tactic rewrites goals using a collection of lemmas. Users can control which definitions to unfold by providing a list of identifiers. This is one of the most powerful tools in `Lean 4` because it automates a lot of routine simplification.

The `omega` tactic. When a goal involves linear arithmetic over natural numbers or integers, the `omega` tactic can often solve it automatically. This work uses `omega` in several proofs to handle arithmetic facts like deriving contradictions from assumptions like $\neg(x \leq y)$.

The `mathlib4` library. `mathlib4` is a large library of formalized mathematics for `Lean 4`. This project mainly uses it for the `List.Perm` relation, which formally represents permutations of lists.

2.2 What metaprogramming means in Lean

Metaprogramming in `Lean 4` means writing `Lean 4` code that manipulates `Lean 4` syntax. There are several levels of metaprogramming in `Lean 4`, but this project focuses on macros.

A macro is a syntax transformation that happens before elaboration. When the `Lean 4` parser sees a macro created, it expands it according to the macro definition. The expanded syntax is then processed as if the user had written it directly.

There are two main kinds of macros used in this project.

Tactic macros. These macros expand into tactic sequences. They let users create custom tactics by combining existing tactics. For example, a tactic macro called `solve_base_case` might expand to `simp [def1, def2]`. Then proofs can use `solve_base_case` instead of typing the full `simp` call every time.

Command macros. These macros expand into top level declarations like definitions or theorems. They allow automatic generation of repetitive declarations. For example, if there are many data structures that all need the same kind of correctness lemma, a command macro can take the names as parameters and generate the theorem statement and proof.

2.3 Syntax quotation and splicing

The key technique for making macros in `Lean 4` is the syntax quotation. When writing `` `(tactic| simp [foo, bar]) ``, the backtick and vertical bar create a piece of quoted syntax. This is not a string but an actual syntax tree that `Lean 4` will later process.

Inside a quotation, arguments can be spliced in using the dollar sign. For example, if there is a macro parameter `h:term`, it can be spliced into the quotation with `$h`. This allows for writing generic templates that work for inputs of any type.

The syntax is explained in detail in the `Lean 4` metaprogramming book and the reference manual [2, 3, 6]. Learning mostly came from reading the examples in the manual and testing.

3 Case study 1: Stacks and queues with macros

The project starts with stacks and queues because they are simple enough where is straightforward but also complex enough to see the benefits of metaprogramming.

3.1 Data structure definitions

A stack is a last in first out data structure. The implementation wraps a list.

Listing 1: Stack definition from `stacks&queues.lean`

```
structure Stack (α : Type u) :=
  (data : List α)
```

```
deriving Repr

namespace Stack

def empty : Stack α := ⟨[]⟩

def push (x : α) (s : Stack α) : Stack α :=
  ⟨x :: s.data⟩

def pop (s : Stack α) : Option (α  Stack α) :=
  match s.data with
  | [] => none
  | x :: xs => some (x, ⟨xs⟩)

end Stack
```

The `push` operation adds an element to the front of the underlying list while the `pop` operation pattern matches on the list. If the list is empty, it returns `none`, but if the list has a head `x` and a tail `xs`, it returns `some (x, ⟨xs⟩)`, which is the popped element paired with the remaining stack.

A queue has almost the same interface except they add elements to the back of the list.

Listing 2: Queue definition with the same structure as Stack

```
structure Queue (α : Type u) :=
  (data : List α)
deriving Repr

namespace Queue

def push (x : α) (q : Queue α) : Queue α :=
  ⟨x :: q.data⟩

def pop (q : Queue α) : Option (α  Queue α) :=
  match q.data with
  | [] => none
  | x :: xs => some (x, ⟨xs⟩)

end Queue
```

3.2 Correctness specification: pop after push

The key correctness property for both structures is that pushing an element and then immediately popping should give back that element and the original structure. For stacks, the theorem is:

Listing 3: Correctness theorem for stacks

```
theorem pop_push (x : α) (s : Stack α) :
  Stack.pop (Stack.push x s) = some (x, s) := by
  simp [Stack.push, Stack.pop]
```

This theorem says that for any type α, any element `x :` α and any stack `s : Stack` α, the following holds. If `x` is pushed onto `s` to get a new stack and then that new stack is popped, the result is `some (x, s)`.

The proof is just one line: `simp [Stack.push, Stack.pop]`. This tells `Lean 4` to simplify the goal by unfolding the definitions of `Stack.push` and `Stack.pop`. Here is what happens step by step.

Step 1. The goal starts as `Stack.pop (Stack.push x s) = some (x, s)`.

Step 2. Unfolding `Stack.push` shows that since `s` has type `Stack` α, which is a structure with one field `data`, the notation `s.data` refers to the underlying list. The definition of `push` is ⟨`x :: s.data`⟩, so the goal becomes `Stack.pop` ⟨`x :: s.data`⟩ `= some (x, s)`.

Step 3. Unfolding `Stack.pop` shows pattern matching on the underlying list. Since the list is `x :: s.data`, the match selects the non-empty branch and returns `some (x,` ⟨`s.data`⟩`)`. The goal becomes `some (x,` ⟨`s.data`⟩`) = some (x, s)`.

Step 4. The expression ⟨`s.data`⟩ is equal to `s` because `Stack` is a single-field structure. So the goal becomes `some (x, s) = some (x, s)`, which is true by reflexivity.

This is an example of a proof that's trivial in concept but requires knowing which definitions to unfold. The `simp` tactic handles all of this automatically.

The queue version is identical.

Listing 4: Correctness theorem for queues

```
theorem queue_pop_push (x : α) (q : Queue α) :
  Queue.pop (Queue.push x q) = some (x, q) := by
  simp [Queue.push, Queue.pop]
```

3.3 Tactic macro: `stacks`

Even though the proof is just one line, the pattern `simp [Stack.push, Stack.pop]` appears in multiple places during development. This is why it makes sense to wrap it in a tactic macro.

Listing 5: The `stacks` tactic macro

```
macro "stacks" : tactic =>
  `(tactic| simp [Stack.push, Stack.pop])
```

The syntax breakdown is as follows. The keyword `macro` starts a macro declaration. The string `"stacks"` is the new syntax being defined. The annotation `: tactic` says this macro expands into a tactic. The right-hand side after `=>` is the expansion. The backtick and `(tactic| ...)` create a syntax quotation for a tactic block. Inside the quotation is the `simp` call to generate.

Now the `pop_push` theorem can be rewritten using the macro.

Listing 6: Using the `stacks` macro

```
theorem pop_push (x : α) (s : Stack α) :
  Stack.pop (Stack.push x s) = some (x, s) := by
  stacks
```

When `Lean 4` sees `stacks` in the proof, it expands it to `simp [Stack.push, Stack.pop]` before elaboration. The proof has exactly the same meaning, but now there is a name for the pattern.

This provides several benefits. First, the proof script is shorter and more readable. Instead of seeing a `simp` call with a list of identifiers, there is a name that describes what the tactic is doing. Second, if the implementation changes or more lemmas need to be added to the `simp` set, only the macro definition needs to be updated. All proofs that use the macro will automatically pick up the change.

3.4 Tactic macro: `queues`

There is an identical macro for queues.

Listing 7: The `queues` tactic macro

```
macro "queues" : tactic =>
  `(tactic| simp [Queue.push, Queue.pop])
```

This can then be used in the queue proof.

Listing 8: Using the `queues` macro

```
theorem queue_pop_push (x : α) (q : Queue α) :
  Queue.pop (Queue.push x q) = some (x, q) := by
  queues
```

It might seem like overkill to write a macro for a one line proof. However, the reason becomes clear at scale. In a real project, there would be many more operations on stacks and queues and many more lemmas. Each lemma might need the same `simp` set to unfold definitions. If there are 20 lemmas that all use `simp [Stack.push, Stack.pop, Stack.empty, Stack.size, ...]` with a long list of identifiers, mistakes will happen. An identifier might be forgotten or a name might be misspelled. The macro puts that list in one place.

3.5 Command macro: `mk_pop_push`

The next level of metaprogramming is to generate entire theorems automatically. The observation is that the `pop_push` theorem for stacks and the `queue_pop_push` theorem for queues have exactly the same structure. They only differ in the names of the type and the operations.

This motivates a command macro that takes those names as parameters and generates the theorem.

Listing 9: The `mk_pop_push` command macro

```
macro "mk_pop_push␣" T:ident push:ident pop:ident thm:ident : command =>
  `(
    theorem $thm {α} (x : α) (s : $T α) :
      $pop ($push x s) = some (x, s) := by
      simp [$push:ident, $pop:ident]
  )
```

This is more complex than the tactic macros, so here is a detailed breakdown.

The first line declares a macro that expects four identifiers as arguments. The first identifier `T` is the type constructor name (like `Stack` or `Queue`). The second identifier `push` is the name of the push operation. The third identifier `pop` is the name of the pop operation. The fourth identifier `thm` is the name to give to the generated theorem.

The annotation `: command` says this macro expands into a top level command, not a tactic.

The right-hand side is a quotation of a theorem declaration. The backtick and parentheses create the quotation. Inside the quotation, dollar signs are used to splice in the macro arguments.

Here is what happens when the macro is invoked.

Listing 10: Invoking the command macro for Stack

```
mk_pop_push Stack Stack.push Stack.pop stack_correct
```

`Lean 4` parses this and sees that it matches the `mk_pop_push` macro syntax. It binds `T` to `Stack`, `push` to `Stack.push`, `pop` to `Stack.pop` and `thm` to `stack_correct`. Then it expands the quotation by substituting these bindings.

The result after expansion is:

Listing 11: Code generated by the macro

```
theorem stack_correct {α} (x : α) (s : Stack α) :
  Stack.pop (Stack.push x s) = some (x, s) := by
  simp [Stack.push, Stack.pop]
```

This is exactly the theorem that is needed. The queue version can be generated the same way.

Listing 12: Invoking the command macro for Queue

```
mk_pop_push Queue Queue.push Queue.pop queue_correct
```

This expands to:

Listing 13: Generated theorem for Queue

```
theorem queue_correct {α} (x : α) (s : Queue α) :
  Queue.pop (Queue.push x s) = some (x, s) := by
  simp [Queue.push, Queue.pop]
```

Two macro invocations have generated two complete correctness theorems.

3.6 Why this matters for large scale verification

This example might seem small but it shows an important principle for verification at scale.

In real software verification project there might be dozens of ADT's (abstract data types) that all follow similar patterns. For example, there might be a stack, a queue, a deque, a priority queue, a hash table and more. Each one needs a family of basic correctness lemmas that relate operations to each other.

If all those lemmas have to be written by hand, it will take a very long time and mistakes are more than likely to occur. Different lemmas might have slightly different statements even when the structures are the same. The proof scripts might use different tactic orders even when the underlying reasoning is identical. This makes the codebase harder to understand and maintain.

With command macros, uniformity can be enforced. The macro encodes the pattern once. Then instances of the pattern can be generated for each data structure. This reduces errors because the pattern only needs to be correct once. It also makes the codebase easier to navigate because the same pattern always looks the same.

This is exactly the same reason functions or classes are used in normal programming. Copying and pasting the same logic 20 times is bad practice. The better approach is to factor it out and reuse it. Metaprogramming lets this be done for proofs and specifications, not just for executable code.

4 Case study 2: Insertion sort with correctness proofs

The second case study is insertion sort. This is a more interesting algorithm than stack and queue operations and the correctness proofs are more detailed. More proof complexity means more opportunities for repetition, which makes metaprogramming more valuable.

4.1 The insertion sort algorithm

Insertion sort is a simple sorting algorithm that builds a sorted list one element at a time. The idea is to insert each element of the input list into the correct position in a growing sorted list.

The implementation is split into two functions. The first function `insert` takes an element and a sorted list and inserts the element into the correct position. The second function `insertionSort` sorts a list by recursively sorting the tail and then inserting the head into the sorted tail.

Listing 14: Insertion sort implementation

```
def insert (x : Nat) : List Nat  List Nat
| [] => [x]
| y :: ys =>
    if x  y then
      x :: y :: ys
    else
      y :: insert x ys

def insertionSort : List Nat  List Nat
| [] => []
| x :: xs => insert x (insertionSort xs)
```

Here is how `insert` works. If the list is empty, the result is just a singleton list containing `x`. If the list has a head `y` and a tail `ys`, then `x` and `y` are compared. If `x` $\leq$ `y`, then `x` belongs at the front, so the result is `x :: y :: ys`. Otherwise, `y` belongs at the front, so `x` is recursively inserted into `ys` and `y` is added to the front of the result.

The function `insertionSort` is simpler. An empty list is already sorted, so it is returned as is. For a nonempty list `x :: xs`, the tail `xs` is recursively sorted to get a sorted list, and then `x` is inserted into that sorted list.

4.2 The sortedness predicate

To talk about correctness, there needs to be a way to express that a list is sorted. The predicate `isSorted` is defined to be true when the list is in nondecreasing order.

Listing 15: The isSorted predicate

```
def isSorted : List Nat  Prop
| [] => True
| [_] => True
| a :: b :: rest => a  b  isSorted (b :: rest)
```

This definition has three cases. An empty list is sorted by definition. A singleton list is sorted by definition. A list with at least two elements `a` and `b` is sorted if `a` $\leq$ `b` and the tail starting from `b` is sorted.

This is a recursive definition that matches the structure of the list. It is convenient for proofs because pattern matching can be done on the list structure and the `isSorted` structure at the same time.

4.3 Correctness theorem 1: the output is sorted

The first correctness theorem says that `insertionSort` produces a sorted list.

Listing 16: Sortedness theorem

```
theorem insertionSort_sorted (xs : List Nat) :
    isSorted (insertionSort xs) := by
  induction xs with
  | nil =>
      simp [insertionSort, isSorted]
  | cons x xs ih =>
      simp [insertionSort]
      exact insert_preserves_sorted (insertionSort xs) x ih
```

The proof is by induction on the list `xs`. In the base case where `xs` is empty, `insertionSort []` simplifies to `[]`, which is sorted by definition. The `simp` tactic handles this automatically.

In the inductive step, `xs = x :: xs'` where `xs'` is the tail. The induction hypothesis `ih` says that `isSorted (insertionSort xs')` is true. By definition, `insertionSort (x :: xs')` expands to `insert x (insertionSort xs')`. So the goal becomes showing that `isSorted (insert x (insertionSort xs'))`.

This follows from a lemma called `insert_preserves_sorted`, which says that inserting an element into a sorted list produces a sorted list. Here is the lemma and its proof.

Listing 17: Key lemma: inserting preserves sortedness

```
lemma insert_preserves_sorted (xs : List Nat) (x : Nat)
    (h_sorted : isSorted xs) : isSorted (insert x xs) := by
  induction xs with
  | nil =>
      simp [insert, isSorted]
  | cons y ys ih =>
      cases ys with
      | nil =>
          by_cases hxy : x  y
           simp [insert, hxy, isSorted]
           simp [insert, hxy, isSorted]
```

```
            omega
      | cons z zs =>
          by_cases hxy : x  y
           simp [insert, hxy, isSorted]
            exact h_sorted
           have h1 : y  z := h_sorted.left
            have h2 : isSorted (z :: zs) := h_sorted.right
            simp [insert, hxy]
            by_cases hxz : x  z
             simp [hxz, isSorted]
              constructor
               omega
               exact h_sorted.right
             simp only [hxz, ite_false, isSorted]
              have ih_result := ih h2
              simp only [insert, hxz, ite_false] at ih_result
              exact ⟨h1, ih_result⟩
```

This proof is much longer because many cases need to be handled. The overall structure is induction on `xs`.

In the base case where `xs` is empty, `insert x []` is just `[x]`, which is sorted.

In the inductive case where `xs = y :: ys`, there is a case split on the shape of `ys`. If `ys` is empty, then `xs = [y]` is a singleton list. There is a split on whether `x ≤ y`. If yes, then `insert x [y]` is `[x, y]`, which is sorted because `x ≤ y`. If no, then `insert x [y]` is `[y, x]`. To show this is sorted, `y ≤ x` is needed, which follows from `¬(x ≤ y)` using the `omega` tactic.

If `ys = z :: zs` so `xs = y :: z :: zs`, there is another split on `x ≤ y`. If yes, then `insert` returns `x :: y :: z :: zs`. To show this is sorted, `x ≤ y` is needed (which was assumed) and `isSorted (y :: z :: zs)` (which is the hypothesis `h_sorted`).

If no, then `insert` returns `y :: insert x (z :: zs)`. To show this is sorted, `y ≤ (head of (insert x (z :: zs)))` and `isSorted (insert x (z :: zs))` are needed. The second part follows from the induction hypothesis. The first part requires another case split on `x ≤ z`, which produces subgoals that are solved by `omega` or by using properties of `h_sorted`.

This proof has a lot of bookkeeping. Assumptions about sortedness need to be tracked, multiple conditionals need to be split and the induction hypothesis needs to be applied at the right place. The `simp` tactic helps with unfolding definitions and the `omega` tactic helps with arithmetic, but the overall structure requires careful case analysis.

4.4 Correctness theorem 2: the output is a permutation

The second correctness theorem says that `insertionSort` returns a permutation of the input. This uses the `List.Perm` relation from `mathlib4`. Intuitively, `xs.Perm ys` means `ys` is a rearrangement of `xs` with the same elements and the same counts.

Listing 18: Permutation theorem

```
theorem insertionSort_perm (xs : List Nat) :
    xs.Perm (insertionSort xs) := by
  induction xs with
  | nil =>
      simp [insertionSort]
  | cons x xs ih =>
      simp [insertionSort]
      apply List.Perm.trans
       exact List.Perm.cons x ih
       exact insert_is_permutation (insertionSort xs) x
```

Again, the proof is by induction. The base case is trivial because the empty list is a permutation of itself.

In the inductive case, the goal is to show `(x :: xs).Perm (insertionSort (x :: xs))`. By definition, `insertionSort (x :: xs)` is `insert x (insertionSort xs)`. So the goal becomes `(x :: xs).Perm (insert x (insertionSort xs))`.

The induction hypothesis says `xs.Perm (insertionSort xs)`. The desired permutation is built in two steps.

Step 1: `(x :: xs).Perm (x :: insertionSort xs)`. This follows from the induction hypothesis by applying `List.Perm.cons`, which says that if `xs.Perm ys` then `(x :: xs).Perm (x :: ys)`.

Step 2: `(x :: insertionSort xs).Perm (insert x (insertionSort xs))`. This follows from a lemma called `insert_is_permutation`.

Finally, these two permutations are chained together using `List.Perm.trans`, which is the transitivity of permutation.

The key lemma is:

Listing 19: Key lemma: inserting preserves permutation

```
lemma insert_is_permutation (xs : List Nat) (x : Nat) :
    (x :: xs).Perm (insert x xs) := by
  induction xs with
  | nil =>
      simp [insert]
  | cons y ys ih =>
      simp only [insert]
      by_cases hxy : x  y
       simp [hxy]
       simp [hxy]
        apply List.Perm.trans
         exact List.Perm.swap y x ys
         exact List.Perm.cons y ih
```

The structure is similar to the sortedness lemma but the reasoning is different. In the base case, `insert x []` is `[x]`, which is trivially a permutation of `[x]`.

In the inductive case where `xs = y :: ys`, there is a split on $x \leq y$. If yes, then `insert x (y :: ys)` is `x :: y :: ys` which is a permutation of `x :: y :: ys`.

If no, then `insert x (y :: ys)` is `y :: insert x ys`. The goal is to show `(x :: y :: ys).Perm (y :: insert x ys)`.

This permutation is built in two steps. First, swap `x` and `y` to get `(x :: y :: ys).Perm (y :: x :: ys)`. This uses the lemma `List.Perm.swap`. Then, apply the cons constructor to the induction hypothesis to get `(y :: x :: ys).Perm (y :: insert x ys)`. Lastly, chain the two permutations with transitivity.

4.5 Brief summary of the direct proof for insertion sort

At this point, `insertionSort` has been proved correct in both senses. It produces a sorted list and it preserves the elements of the input.

However, the proofs are long and they have repeated patterns. The sortedness proof does multiple case splits on list shapes and on comparisons and uses `simp` and `omega` in similar ways. The permutation proof uses the same "swap then cons then chain" structure in multiple places.

This is where metaprogramming becomes valuable.

5 Metaprogramming in the insertion sort proofs

After finishing the direct proofs, a second namespace called `MacroVersion` was created. In this namespace, the same theorems are proved but custom macros are used to handle the repetitive parts. The theorems have the same statements and the same logical content, but the proof scripts are shorter and more readable.

5.1 Tactic macro: `solve_nil_sorted`

The first macro is for the base case of the sortedness proofs. When the list is empty, the proof is always the same: simplify using the definitions of `insert` and `isSorted`.

Listing 20: Macro for the empty list case

```
macro "solve_nil_sorted" : tactic =>
  `(tactic| simp [InsertionSort.insert, InsertionSort.isSorted])
```

This is similar to the `stacks` macro from earlier. It expands into a `simp` call with a fixed list of definitions.

This macro is used in the base case of `insert_sorted_macro`.

Listing 21: Using the macro in a proof

```
lemma insert_sorted_macro (xs : List Nat) (x : Nat)
    (h_sorted : InsertionSort.isSorted xs) :
    InsertionSort.isSorted (InsertionSort.insert x xs) := by
  induction xs with
  | nil => solve_nil_sorted
  | cons y ys ih => ...
```

When `Lean 4` sees `solve_nil_sorted`, it expands it to the `simp` call. The proof is the same as before, but now there is a name for the pattern.

5.2 Tactic macro: `chain_perms`

The permutation proofs use transitivity several times to chain together permutation facts. The pattern is always the same: `apply List.Perm.trans` followed by two subgoals that provide the two permutations to chain.

A macro packages this pattern.

Listing 22: Macro for chaining permutations

```
macro "chain_perms" h1:term "with" h2:term : tactic =>
  `(tactic| exact List.Perm.trans $h1 $h2)
```

This macro takes two arguments. Both have type `term`, which means they can be any `Lean 4` expression. The macro expands into `exact List.Perm.trans $h1 $h2`, where the dollar signs splice in the arguments.

This macro is used in the permutation proofs.

Listing 23: Using the `chain_perms` macro

```
theorem sort_perm_macro (xs : List Nat) :
    xs.Perm (InsertionSort.insertionSort xs) := by
  induction xs with
  | nil => simp [InsertionSort.insertionSort]
  | cons x xs ih =>
      simp [InsertionSort.insertionSort]
      have cons_perm := List.Perm.cons x ih
      have insert_perm := insert_perm_macro (insertionSort xs) x
      chain_perms cons_perm with insert_perm
```

In the last line, `chain_perms cons_perm with insert_perm` expands to `exact List.Perm.trans cons_perm insert_perm`. The proof is the same as before, but the syntax is cleaner.

Why is this better? First, the macro makes the intent clear. When reading `chain_perms`, it is immediately obvious what is happening, without having to parse the `exact` and `List.Perm.trans` syntax. Second, if the way transitivity is handled needs to change later (maybe using a different tactic or adding some logging), only the macro definition needs to be changed.

5.3 Tactic macro: `swap_and_cons`

The most interesting macro is `swap_and_cons`. This packages a common two step argument in permutation proofs.

Listing 24: The `swap_and_cons` macro

```
macro "swap_and_cons" y:term x:term ys:term ih:term : tactic =>
  `(tactic| (
    have swap := List.Perm.swap $y $x $ys;
    have cons_ih := List.Perm.cons $y $ih;
    exact List.Perm.trans swap cons_ih
  ))
```

This macro takes four arguments: the variables `y`, `x`, `ys` and `ih`. It expands into a block that first builds the swap permutation, then builds the cons permutation, then chains them together.

Here is how this is used in context. In the `insert_perm_macro` lemma, the difficult case is when `¬(x ≤ y)`. The goal is to show `(x :: y :: ys).Perm (y :: insert x ys)`.

Without the macro, the proof looks like this:

Listing 25: Proof without the macro

```
have swap := List.Perm.swap y x ys
have cons_ih := List.Perm.cons y ih
exact List.Perm.trans swap cons_ih
```

With the macro, the proof is:

Listing 26: Proof with the macro

```
swap_and_cons y x ys ih
```

The macro does not make the proof much shorter, but it makes it much more readable. When seeing `swap_and_cons`, the structure of the argument is immediately clear, without having to parse three lines of tactic script.

More importantly, the macro enforces consistency. If there are 10 different lemmas that all use this swap and cons pattern, they all do it the same way. If a better way to structure the argument is discovered later, the macro can be updated and all 10 lemmas will automatically use the new approach.

5.4 Macro for single element lists: `solve_single_sorted`

The last macro is `solve_single_sorted`. This handles a specific case in the sortedness proof where the list has one element and there needs to be a split on `x ≤ y`.

Listing 27: The `solve_single_sorted` macro

```
macro "solve_single_sorted" h:term : tactic =>
  `(tactic| (
    simp only [InsertionSort.insert];
    split;
     simp [InsertionSort.isSorted];
      exact $h;
     simp [InsertionSort.isSorted];
      have : (x  y) := by assumption;
      omega
  ))
```

This is a more complex macro because it generates multiple tactic steps. It first simplifies to unfold `insert`, splits on the conditional and then handles each branch separately.

This macro was not used in the final version of the proofs because the cases did not line up perfectly with the macro structure. But it is included here to show how more complex proof patterns can be packaged into macros.

6 Why metaprogramming is essential for software verification

Based on this project, metaprogramming is not just a convenience but is actually essential for practical verification. Here is why.

6.1 The scaling problem

Verification does not scale the same way as normal programming. In normal programming, if there are 100 functions, 100 implementations are needed and that's it. In verification, if there are 100 functions, 100 implementations are needed plus hundreds of lemmas that relate them.

For example, if there is a data structure with 10 operations, the project might need:

- 10 lemmas that show each operation respects the invariant
- 45 lemmas that show how pairs of operations interact (since there are $\binom{10}{2} = 45$ pairs)
- additional lemmas for special properties

This is a huge amount of proof code so writing it all by hand is not realistic and that's why macros and tactics are needed.

6.2 The repetition problem

Many of these lemmas have similar structure. For example, every operation on a stack might need a lemma that says "if the stack satisfies the invariant before the operation, it satisfies the invariant after the operation". The statements of these lemmas are similar and the proofs are often similar too.

Without metaprogramming, each proof has to be written by hand, even when they are almost identical. This is tedious and error prone. A proof might be copied and pasted and a variable name might not be changed, or tactics might be applied in a slightly different order, introducing an inconsistency.

With metaprogramming, common patterns can be factored out. Tactic macros package common proof steps. Command macros generate families of similar theorems automatically. This reduces errors and makes the codebase more maintainable.

6.3 The maintainability problem

Verification projects evolve over time. A definition might be changed to make it more efficient or more general. When a definition is changed, proofs that depend on it often break.

If the proofs are written in low level tactics then fixing them can be very tedious. Each proof has to be read carefully to figure out which steps need to change. This becomes a huge burden and turns into "busy work" especially if the same pattern appears in many proofs because the same fix has to be made many times.

If the proofs use macros, fixing them is often much easier. The macro definition can be updated and all the proofs that use the macro automatically pick up the change. This is the same benefit as using functions instead of copy pasting code in normal programming.

6.4 Readability and error prevention

Proof scripts in theorem provers can become long and difficult to read, often consisting of many low-level tactic invocations where the high-level structure is hard to see. Macros introduce abstraction boundaries: instead of dozens of lines of tactics, a proof can be written as a few descriptive macro calls, making scripts easier to understand, review and maintain.

Macros also reduce errors caused by repeating the same pattern by hand (missing hypotheses, wrong order, wrong names), which can be difficult to debug given cryptic checker messages. Once a pattern is encoded correctly in a macro, every use reuses the same structure. While a buggy macro can propagate mistakes, macros are typically small and self contained, making them easier to test and fix than large tactic scripts.

7 Challenges faced from this project

Understanding syntax quotation. At first, the syntax for quotations and splicing was confusing. The backtick and dollar sign notation felt weird. It took some experimentation to understand that quotations are syntax trees, not strings, and that splicing inserts syntax into syntax.

Getting the macro types right. Sometimes a macro that was supposed to be a tactic macro was accidentally made a term macro or a command macro. The error messages were not always helpful. Learning to pay close attention to the annotation after the macro name (like `: tactic` or `: command`) was important.

Debugging macro expansions. When a macro does not work, it can be hard to see what went wrong. Learning to use the `#check` command and to manually expand the macro on paper to see what code it generates was helpful.

Insertion sort proof complexity. The insertion sort proofs were significantly more challenging than the stack and queue proofs. The `insert_preserves_sorted` lemma required careful case analysis on multiple levels (empty lists, singleton lists, lists with two or more elements) and nested case splits on comparison conditions. Getting the induction hypothesis to apply correctly in the right cases took several attempts. The permutation proofs also required understanding how to compose permutation lemmas using `List.Perm.swap`, `List.Perm.cons` and `List.Perm.trans` in the correct order, which was not immediately obvious.

Scope limitations and future work. The project initially planned to include palindromes as a third case study, which would have demonstrated metaprogramming with recursive predicates and structural induction on more complex patterns. However, due to time constraints and the complexity encountered with insertion sort, the palindrome verification was not completed. This would have been the natural next step, exploring how macros could handle bidirectional reasoning (proving that a list equals its reverse).

8 Conclusion

The most important insight from this project is that writing verified software is fundamentally a software engineering challenge, not just a logical one. While proving correctness requires mathematical rigor, the practical barrier to verification is managing the scale and complexity of proof code. Metaprogramming in `Lean 4` addresses this by treating proofs as first-class code that can be abstracted, refactored and reused. The experience of building macros for these data structures and algorithms showed that even in a small project, proof patterns repeat enough to justify automation for verifying software code. The gap between these sample examples and production verification is not about harder logic but about engineering discipline: organizing proof code, maintaining consistency and building the right abstractions.

References

[1] Lean Prover Contributors. *Functional Programming in Lean: Insertion Sort and Array Mutation.* `https://leanprover.github.io/functional_programming_in_lean/programs-proofs/insertion-sort.html`

[2] Lean Prover Community. *The Lean 4 Metaprogramming Book.* `https://leanprover-community.github.io/lean4-metaprogramming-book/`

[3] Lean Prover Community. *The Lean 4 Metaprogramming Book: Macros.* `https://leanprover-community.github.io/lean4-metaprogramming-book/main/06_macros.html`

[4] James Oswald. *Insertion sort in Lean 4.* `https://jamesoswald.dev/posts/lean4-insertion-sort/`

[5] Brown CS1951x course staff. *The Hitchhiker's Guide to Logical Verification.* `https://browncs1951x.github.io/static/files/hitchhikersguide.pdf`

[6] Lean Prover Contributors. *Lean Reference: Notations and Macros, Macros.* `https://lean-lang.org/doc/reference/latest/Notations-and-Macros/Macros/`