ComputoPermuto: A Practical Guide to JSON Transformations

Contents

Contents

1 Computo/Permuto: A Practical Guide to JSON Transformations

A comprehensive guide to mastering JSON transformation with Computo and Permuto

---

1.1 Table of Contents

1.1.1 Part I: Foundations

1.1.2 Part II: Core Operations

1.1.3 Part III: Advanced Techniques

1.1.4 Part IV: Production Usage

Chapter 14: Best Practices and Patterns - Code organization and reusability - Testing transformation scripts - Maintainable patterns

Chapter 15: Multiple Input Processing and JSON Patch Operations - Working with multiple input documents using `$inputs` - RFC 6902 JSON Patch support with `diff` and `patch` operators - Document versioning and change management - Multi-document processing patterns

1.1.5 Appendices

Appendix A: Complete Operator Reference - Comprehensive reference for all 25 operators - Syntax examples and use cases - Quick lookup guide

---

1.2 Quick Navigation

1.2.1 For Beginners

Start with Chapter 1 to understand the motivation, then proceed through Chapter 2 for setup and Chapter 3 for fundamentals.

1.2.2 For Experienced Developers

Skip directly to Chapter 3 for syntax basics, then jump to specific topics of interest or the Operator Reference for quick lookup.

1.2.3 For Specific Use Cases

- **API Integration**: Chapter 11
- **Configuration Management**: Chapter 9
- **Data Pipelines**: Chapter 10
- **Document Versioning**: Chapter 15
- **Performance Tuning**: Chapter 12

1.2.4 Reference Materials

- **Complete Operator List**: Appendix A
- **Error Types**: Chapter 13
- **Best Practices**: Chapter 14

1.3 About This Guide

This guide assumes you are an experienced developer familiar with JSON, functional programming concepts, and command-line tools. We focus on practical examples and real-world scenarios rather than theoretical explanations.

1.3.1 What You'll Learn

- Master both Computo (programmatic logic) and Permuto (declarative templates)
- Build robust JSON transformation pipelines

- Handle complex multi-document processing scenarios
- Implement RFC 6902 JSON Patch workflows
- Apply best practices for production usage

1.3.2 Prerequisites

- Familiarity with JSON syntax and structure
- Basic understanding of functional programming (map, filter, reduce)
- Command-line experience
- C++ build tools (for building from source)

Ready to transform JSON like a pro? Let's begin with Chapter 1: Why JSON Transformation Matters.

2 Computo/Permuto: A Practical Guide to JSON Transformations

2.1 Chapter 1: Why JSON Transformation Matters

Welcome. If you're reading this, you're likely an experienced developer. You live and breathe APIs, configuration files, and data interchange. You know that JSON isn't just a data format; it's the lingua franca of the modern web. You also know that while creating and parsing JSON is a solved problem, **transforming it** can be surprisingly messy.

That's the gap this guide—and the Computo/Permuto toolkit—is designed to fill. We won't waste your time explaining what a JSON object is or how a `map` function works. Instead, we'll get straight to the point: giving you a powerful, safe, and elegant way to reshape JSON from one form to another.

2.1.1 The Problem Space: The Daily Grind of Reshaping JSON

Think about the last time you had to manipulate a JSON structure. Did it feel like one of these scenarios?

- **API Aggregation:** You're building a backend-for-frontend (BFF) and need to call three different microservices. Each returns a JSON object with a slightly different structure. Your job is to pick and choose fields from each, combine them into a single, clean JSON response, and send it to the client.
- **Configuration Management:** You have a master configuration template for your application. You need to generate specific `config.json` files for your `development`, `staging`, and `production` environments, each with different database credentials, feature flags, and logging levels.
- **Data Normalization:** You're ingesting data from a third-party API. The data is messy: some keys are camelCase while others are snake_case, numbers are sometimes strings, and optional fields are sometimes `null` and other times omitted entirely. You need to sanitize and normalize this data into a consistent format for your own systems.
- **Dynamic Payloads:** You're integrating with multiple Large Language Model (LLM) providers. OpenAI wants a payload with a `messages` array, while Anthropic wants a single `prompt` string. You need to build these different JSON payloads from a single, canonical request object within your application.

Traditionally, you'd solve these problems by writing imperative code in your language of choice—Python, JavaScript, Go, C++. You'd loop through arrays, check for the existence of keys, and manually build new objects. It works, but it's often verbose, error-prone, and mixes transformation logic deep within your business logic.

2.1.2 Introducing the Computo/Permuto Solution

Computo and Permuto offer a better way. They form a powerful, two-layer system designed specifically for JSON-to-JSON transformation, where each layer has a distinct responsibility.

1. **Permuto: The Declarative Templating Engine** Think of Permuto as a smart, structure-aware "mail merge" for JSON. You provide it a template, and it fills in the blanks. It's for simple, declarative substitutions. If you just need to map values from a context object into a new structure, Permuto is your tool. It's fast, simple, and safe.

A quick taste: `{"user_id": "${/user/id}"}`

2. **Computo: The Programmatic Logic Engine** Think of Computo as a safe, sandboxed Lisp or spreadsheet formula engine that lives inside JSON. It provides the programmatic logic that simple templating lacks: conditionals (`if`), iteration (`map`, `filter`, `reduce`), calculations (`+`, `*`), and variable bindings (`let`).

   A quick taste: `["if", <condition>, <then_expr>, <else_expr>]`

2.1.3 The "Aha!" Moment: Logic Driving Templates

The real power emerges when you combine them. You use Computo's logic to decide *how* and *if* to apply Permuto's templates.

Imagine our user summary problem again. If a user is active, we want to generate a full profile. If not, we want a simple error object.

- **Computo's `if` operator** handles the conditional logic.
- It checks a field, like `user.active`.
- If `true`, it calls **Permuto** with a `user_profile_template.json` to generate the rich object.
- If `false`, it constructs a simple `{"error": "User is inactive"}` object.

The script for this transformation is a clean piece of data (a JSON file) that you can store, version, and execute safely, without intermingling complex string manipulation or loops in your core application code.

2.1.4 What's Ahead

In the following chapters, we will explore this toolkit from the ground up.

- We'll start by **setting up your environment**.
- Then, we'll master the fundamentals of **Computo and Permuto individually**.
- Finally, we will combine them to build **sophisticated data pipelines** that solve real-world problems like the ones mentioned above.

By the end of this guide, you'll have a new, powerful tool in your arsenal for one of the most common tasks in modern software development. Let's get started.

2.2 Chapter 2: Setting Up Your Environment

Before we dive into writing transformations, let's get the `computo` command-line interface (CLI) tool built and running on your local machine. Because Computo depends on the Permuto library, we will follow a two-stage process: first building and installing Permuto, then building Computo itself.

This guide uses a standard C++/CMake build process that will be familiar to most developers.

2.2.1 Prerequisites

You will need a C++17 compatible compiler, CMake, and Git. The development headers for the `nlohmann/json` library are also required.

- **Compiler:** GCC, Clang, or MSVC.
- **CMake:** Version 3.15 or later.
- **Git:** For cloning the repositories.
- **Dependencies:** `nlohmann/json` development files.

On a Debian/Ubuntu-based system, you can install the necessary dependencies with:

```
sudo apt-get update
sudo apt-get install build-essential cmake git nlohmann-json3-dev
```

2.2.2 Step 1: Build and Install Permuto

Computo uses Permuto for its powerful templating capabilities, so we must install it first.

2.2.2.1 1a. Clone the Permuto Repository

```
git clone https://github.com/your-username/permuto.git
cd permuto
```

(Note: Replace `your-username/permuto.git` with the actual repository URL.)

2.2.2.2 1b. Configure and Build We'll create a `Release` build for optimal performance.

```
cmake -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build
```

This will compile the Permuto library and its own CLI tool.

2.2.2.3 1c. Install Permuto Now, install the library to a standard system location (typically `/usr/local/lib` and `/usr/local/include`). This step makes it discoverable by other projects, like Computo.

```
# From within the 'permuto' directory
sudo cmake --install build
```

With Permuto installed, your system is now ready to build Computo.

2.2.3 Step 2: Build Computo

Now we'll repeat a similar process for the `computo` project.

2.2.3.1 2a. Clone the Computo Repository

If you are still in the `permuto` directory, navigate out of it first (`cd ..`).

```
git clone https://github.com/your-username/computo.git
cd computo
```

2.2.3.2 2b. Configure and Build

CMake will now automatically find the Permuto library and headers you installed in the previous step.

```
cmake -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build
```

This command compiles the Computo library and the `computo` CLI tool. The final executable will be placed in the `build/` directory.

2.2.3.3 2c. Install Computo

For convenience, install the `computo` CLI tool to a standard system location so you can run it from anywhere without specifying the full path.

```
# From within the 'computo' directory
sudo cmake --install build
```

Alternatively, for a stripped (smaller) binary:

```
sudo cmake --install build --strip
```

This installs the `computo` executable to `/usr/local/bin` (or your system's standard binary location), making it available system-wide.

2.2.4 Step 3: Verify the Installation

You can verify that everything works by running the CLI tool or its test suite.

To check the CLI: If you installed computo system-wide, you can now run it from anywhere. Try running it with no arguments to see the usage message.

```
computo
```

If you haven't installed it yet, you can still test it from the build directory:

```
./build/computo
```

To run the test suite: The project includes a comprehensive set of tests. Running them confirms that Computo, Permuto, and all operators are working correctly together on your system.

```
# From within the 'computo' directory
cd build
ctest --verbose
```

You should see output indicating that all 153 tests have passed.

2.2.5 Your Workspace

You are now ready to start writing transformations. Your typical workflow will involve:

1. Creating a `script.json` file containing your Computo logic.

2. Creating an `input.json` file with the data you want to transform.
3. Running the transformation from anywhere on your system (assuming you installed computo):

```
# Basic usage (compact output)
computo script.json input.json

# Pretty-printed output (recommended for development)
computo --pretty=2 script.json input.json
```

In the next chapter, we will write our first Computo script and explore its fundamental concepts.

2.3 Chapter 3: Computo Basics - Data and Logic

Now that you have a working environment, it's time to start transforming JSON. In this chapter, we'll focus on the core of Computo: its syntax, its fundamental operators, and how it handles data.

At its heart, Computo treats **code as data**. Every Computo script is itself a valid JSON document. This is a powerful concept borrowed from Lisp-like languages that makes scripts easy to generate, store, and even manipulate programmatically.

2.3.1 The Anatomy of a Computo Expression

An "operation" in Computo is represented by a JSON array where the first element is a string identifying the operator.

```
["+", 10, 5]
```

This is a Computo expression that instructs the engine to use the `+` operator on the numbers `10` and `5`. If you were to run this script, the engine would evaluate it and return the JSON number `15`.

Any JSON that isn't an operator call is treated as a **literal value**.

- `42` evaluates to the number `42`.
- `"hello"` evaluates to the string `"hello"`.
- `{"name": "Alice"}` evaluates to a JSON object.

The magic happens when you start nesting these expressions.

2.3.2 Your First Transformation: Simple Arithmetic

Let's start with the simplest possible script.

1. Create a file named `add.json`:

   ```
   ["+", 100, 50]
   ```

2. Create an empty file for our input, `input.json`: *(Computo requires an input file, even if the script doesn't use it yet.)*

   ```
   {}
   ```

3. Now, run it from your terminal:

   ```
   computo add.json input.json
   ```

The output will be:

```
150
```

By default, Computo outputs JSON in compact form. For development and learning, you might prefer pretty-printed output:

```
computo --pretty=2 add.json input.json
```

This produces the same result but with proper indentation when dealing with complex structures.

2.3.3 Understanding Output Formatting

The `--pretty=N` option controls how Computo formats its JSON output:

- **No flag (default)**: Compact output - `{"name":"Alice","age":30}`
- **`--pretty=2`**: Pretty-printed with 2-space indentation
- **`--pretty=4`**: Pretty-printed with 4-space indentation

For learning and development, we recommend using `--pretty=2` to make complex JSON structures easier to read. In production scripts or pipelines, you might prefer the compact default format for efficiency.

2.3.4 Adding Comments to Scripts (CLI Only)

When working with complex transformations, you might want to add comments to your script files for documentation. The CLI tool supports this with the `--comments` flag:

```
# Enable comment parsing in script files
computo --comments --pretty=2 script_with_comments.json input.json
```

Important: Comments are only supported in **script files**, not input files, and only when using the CLI tool (not the C++ library API).

Example commented script:

```
[
  // This creates a user profile object
  "obj", // obj creates a JSON object
  /* Extract the user's name from input data
     and format it for display */
  ["name", ["get", ["$input"], "/user/name"]],
  ["greeting", "Welcome!"] // Static greeting message
]
```

Supported comment styles: - `// Single line comments` - `/* Multi-line comments */`

The Computo engine evaluated the expression and printed the resulting JSON to your console. Now let's try nesting. Change `add.json` to:

```
["+", 100, ["*", 5, 10]]
```

Run it again. The output is `150`. The engine first evaluated the inner expression `["*", 5, 10]` to get `50`, then evaluated the outer expression `["+", 100, 50]`.

2.3.5 Creating Objects and Arrays

After experimenting with basic arithmetic, you might wonder how to create more complex JSON structures. Computo provides specific syntax for constructing objects and arrays.

2.3.5.1 Creating Objects with `obj`

To create a JSON object, use the `obj` operator with key-value pairs:

```
["obj", ["key1", "value1"], ["key2", "value2"]]
```

Let's try this. Create a script called `make_object.json`:

```
["obj",
  ["name", "Alice"],
  ["age", 30],
  ["score", ["+", 85, 15]]
]
```

Run it:

```
computo --pretty=2 make_object.json input.json
```

Output:

```
{
  "name": "Alice",
  "age": 30,
  "score": 100
}
```

Note: We're using `--pretty=2` to format the output with 2-space indentation for readability. Without this flag, the output would be compact: `{"name":"Alice","age":30,"score":100}`

Notice how the `score` value is computed from the nested arithmetic expression `["+", 85, 15]`.

2.3.5.2 Creating Arrays

For arrays, use the special `{"array": [...]}` syntax:

```
{"array": [1, 2, ["*", 3, 4]]}
```

Create a script called `make_array.json`:

```
{"array": [
  "hello",
  42,
  ["+", 10, 5],
  ["obj", ["type", "computed"], ["value", true]]
]}
```

Run it:

```
computo --pretty=2 make_array.json input.json
```

Output:

```
[
  "hello",
  42,
  15,
  {
    "type": "computed",
    "value": true
  }
]
```

This array contains: - A literal string `"hello"` - A literal number `42` - A computed number `15` from `["+", 10, 5]` - A computed object from the `obj` operator

2.3.6 A Note on Syntax: Why `{"array": [...]}`?

You may wonder why literal arrays are written as `{"array": [...]}` instead of a simpler `["arr", ...]` operator. This was a deliberate design choice for one critical reason: **to eliminate ambiguity**.

In Computo, a JSON array like `["operator", ...]` is always an action. But what about a simple array of strings, like `["red", "green", "blue"]`? Without a special rule, the interpreter would see `"red"` and think it's an invalid operator.

The `{"array": [...]}` syntax provides a clear, unambiguous signal to the interpreter: "Treat this as a literal piece of data, not an action to perform."

This allows the core rule—**arrays are for actions, objects are for data**—to hold true, making the language robust and predictable. It trades a small amount of verbosity for absolute clarity.

Now that you know how to construct basic data structures, let's learn how to work with dynamic input data.

2.3.7 Accessing Input Data with `$input` and `get`

Static scripts are only so useful. The real power comes from transforming dynamic input data. Computo makes the entire contents of your `input.json` file available through a special, zero-argument operator: `$input`. Please note that the name of the operator has nothing to do with the filename.

Let's create a more realistic scenario.

1. Update `input.json` with some user data:

   ```
   {
     "user": {
       "id": "u-123",
       "name": "Alice",
       "plan": {
         "name": "premium",
         "monthly_cost": 20
       }
     },
     "last_login_ts": 1672531200
   }
   ```

2. Create a new script, `get_plan.json`, to extract the plan name. To do this, we need another operator: `get`.

   `["get", <object>, <json_pointer>]`

   The `get` operator takes an object to query and a JSON Pointer string to specify which value to extract.

   ```
   ["get", ["$input"], "/user/plan/name"]
   ```

3. Run the script:

   ```
   computo get_plan.json input.json
   ```

The output will be:

```
"premium"
```

For simple values like strings, the `--pretty` flag doesn't make much difference, but it's helpful for complex structures. The engine first evaluated `["$input"]` to get the entire input object, then `get` extracted the value at `/user/plan/name`.

2.3.8 Storing Intermediate Values with `let` and `$`

Often, you need to calculate an intermediate value and reuse it multiple times. Constantly re-evaluating long expressions like `["get", ["$input"], "/user/plan"]` would be tedious and inefficient.

Computo provides the `let` operator for binding values to variables within a specific scope.

```
["let", [["var1", <expr1>], ...], <body_expr>]
```

Variables defined in a `let` block are accessed using the `$` operator: `["$", "/var_name"]`. Note the required `/` prefix, which makes variable access look similar to root-level JSON Pointers.

Let's build a new user object containing just the name and monthly cost.

1. Create a script named `build_user.json`:

```
["let",
  [
    ["user_plan", ["get", ["$input"], "/user/plan"]],
    ["user_name", ["get", ["$input"], "/user/name"]]
  ],
  ["obj",
    ["name", ["$", "/user_name"]],
    ["cost", ["get", ["$", "/user_plan"], "/monthly_cost"]]
  ]
]
```

2. Run the script with our previous `input.json`:

```
computo --pretty=2 build_user.json input.json
```

The output is a completely new JSON object, built from pieces of the input:

```
{
  "cost": 20,
  "name": "Alice"
}
```

Here's the flow: 1. `let` creates a new scope. 2. It evaluates `["get", ["$input"], "/user/plan"]` and binds the resulting `{"name": "premium", "monthly_cost": 20}` object to the variable `user_plan`. 3. It evaluates `["get", ["$input"], "/user/name"]` and binds the string `"Alice"` to the variable `user_name`. 4. It then evaluates the body expression, substituting `["$", "/user_name"]` with `"Alice"` and using `get` on the `user_plan` object.

2.3.9 In This Chapter

You've learned the fundamental building blocks of Computo: * The **syntax** of operators `["op", ...]` and literal values. * How to **create objects** using the `obj` operator with key-value pairs. * How to **create arrays** using the `{"array": [...]}` syntax. * How to **format output** using the `--pretty=N` CLI option for readable JSON. * How to **add comments** to script files using the `--comments` CLI flag. * How to access the entire input document with **`$input`**. * How to extract specific data with **`get`** and JSON Pointers. * How to create temporary variables with **`let`** and access them with **`$`**.

These fundamental operators give you the power to construct new JSON structures and perform data extraction and reshaping. In the next chapter, we'll switch gears and look at Permuto, the templating engine that complements Computo's logic.

2.4 Chapter 4: Permuto Basics - Templates

In the last chapter, we used Computo's `obj` operator to manually construct a new JSON object piece by piece. This is perfect for when you need fine-grained logical control over every field. But often, your goal is simpler: you have a well-defined output structure and you just need to "fill in the blanks" with data from an input object.

This is a job for **templating**, and it's where **Permuto** shines.

Permuto is Computo's declarative partner. While Computo handles programmatic logic (`if`, `map`, etc.), Permuto handles declarative data shaping. Think of it as a smart, structure-aware "mail merge" for JSON.

2.4.1 The Core Idea: Template + Context = Result

The Permuto workflow is simple and always involves two pieces of JSON:

1. **The Template:** A JSON document that defines the desired output structure. It contains placeholders for where the dynamic data should go.
2. **The Context:** A JSON document that provides the data to fill in the placeholders.

Permuto takes the template, substitutes the values from the context, and produces the final result.

2.4.2 Placeholders and JSON Pointers

Permuto's placeholders use a simple `${...}` syntax. Inside the curly braces, you put the exact same **JSON Pointer** path that you used with Computo's `get` operator.

Let's revisit the user example from the previous chapter.

1. First, create the **context** data. This is the same `input.json` we've been using, which provides the source data.

   `input.json`:

   ```
   {
     "user": {
       "id": "u-123",
       "name": "Alice",
       "plan": {
         "name": "premium",
         "monthly_cost": 20
       }
     },
     "last_login_ts": 1672531200
   }
   ```

2. Next, create the **template** file. This file defines the shape of our desired output.

   `user_template.json`:

   ```
   {
     "profile_id": "${/user/id}",
     "full_name": "${/user/name}",
   ```

```
  "plan_type": "${/user/plan/name}"
}
```

Notice this template is just a plain JSON file. There are no operators, no arrays-as-actions. It's just a passive structure.

2.4.3 Running a Permuto Template with `permuto.apply`

To use a Permuto template from within our Computo-driven workflow, we use the `permuto.apply` operator. This is the bridge between the two systems.

`["permuto.apply", <template>, <context>]`

1. Create a script that uses this operator. Let's call it `apply_template.json`. It will use the `user_template.json` as the template and our `input.json` as the context.

   `apply_template.json`:

   ```
   ["permuto.apply",
     {
       "profile_id": "${/user/id}",
       "full_name": "${/user/name}",
       "plan_type": "${/user/plan/name}"
     },
     ["$input"]
   ]
   ```

   Note: For simplicity, we've embedded the template directly in our script. We could also load it from a file if it were more complex.

2. Now, run the script:

   ```
   computo apply_template.json input.json
   ```

The output is a new JSON object, perfectly matching the template's structure:

```
{
  "full_name": "Alice",
  "plan_type": "premium",
  "profile_id": "u-123"
}
```

This is significantly more concise and readable than building the same object with `["obj", ["profile_id", ["get", ...]], ...]`.

2.4.4 Type Preservation

A key feature of Permuto is that it preserves data types. If a placeholder resolves to a number, boolean, or even a complex object or array, that's what gets inserted into the final result.

Consider this template:

```
{
  "plan_details": "${/user/plan}",
```

```
  "cost": "${/user/plan/monthly_cost}"
}
```

When applied to our `input.json`, this would produce:

```
{
  "cost": 20,
  "plan_details": {
    "name": "premium",
    "monthly_cost": 20
  }
}
```

The `plan_details` key was populated with the entire plan sub-object, and `cost` was populated with the JSON number `20`, not the string `"20"`.

2.4.5 String Interpolation

Permuto can also substitute placeholders inside of strings, a feature known as string interpolation. **This feature is disabled by default** and must be explicitly enabled with a command-line flag.

1. Create a new template, `greeting_template.json`: `json { "message": "Hello ${/user/name}! Your plan is ${/user/plan/name}." }`
2. Create a new script, `apply_greeting.json`, using this template: `json ["permuto.apply", { "message": "Hello ${/user/name}! Your plan is ${/user/plan/name}." }, ["$input"] ]`
3. Run it with the `--interpolation` flag:

   ```
   computo --interpolation apply_greeting.json input.json
   ```

The output shows the values correctly substituted within the string:

```
{
  "message": "Hello Alice! Your plan is premium."
}
```

2.4.6 In This Chapter

You've now seen the other half of the toolkit. You've learned: * How Permuto uses a **template** and a **context** to produce a result. * How to define placeholders using the **`${/json/pointer}`** syntax. * How to invoke Permuto from a Computo script using the **`permuto.apply`** operator. * How to enable **string interpolation** for more complex string construction.

You now have a clear understanding of the two distinct tools at your disposal:

- **Computo:** For programmatic logic, calculations, and step-by-step construction.
- **Permuto:** For declarative, whole-structure templating.

In the next part of this guide, we will dive deeper into Computo's more advanced features, starting with control flow and conditionals. You will see how to use Computo's logic to make intelligent decisions about which templates to apply and how to build the context for them.

2.5 Chapter 5: Control Flow and Conditionals

So far, our scripts have been linear. They execute from the inside out, following a single, predetermined path. To build truly dynamic transformations, we need the ability to make decisions. We need to be able to say: "If this condition is met, do this; otherwise, do that."

This is the role of the `if` operator in Computo.

2.5.1 The `if` Operator

The `if` operator is the primary tool for conditional logic in Computo. Its structure is simple and should be familiar from other programming languages.

`["if", <condition>, <then_expression>, <else_expression>]`

The engine evaluates the `<condition>` expression first. * If the result is "truthy", the engine evaluates and returns the result of the `<then_expression>`. * If the result is "falsy", the engine evaluates and returns the result of the `<else_expression>`.

A crucial feature of `if` is **lazy evaluation**: only the branch that is chosen gets evaluated. The other branch is completely ignored, which is important for both performance and preventing errors.

2.5.2 Computo's Definition of "Truthy" and "Falsy"

Since JSON has several data types, Computo has clear rules for what it considers `true` or `false` in a conditional context. These rules are very similar to those in languages like Python or JavaScript.

Value Type	Falsy (evaluates to `false`)	Truthy (evaluates to `true`)
Boolean	`false`	`true`
Number	`0` and `0.0`	Any non-zero number
String	The empty string `""`	Any non-empty string
Object	The empty object `{}`	Any object with one or more keys
Array	The empty array `[]` or `{"array": []}`	Any array with one or more elements
Null	`null`	(never truthy)

2.5.3 Example: Conditional Greeting

Let's build a script that generates a different output based on whether a user is active.

1. We'll start with a new `input.json` containing a list of users.

   `users_input.json`:

   ```
   {
     "users": [
       { "name": "Alice", "active": true, "plan": "premium" },
       { "name": "Bob", "active": false, "plan": "basic" },
       { "name": "Charlie", "active": true, "plan": "basic" }
     ]
   }
   ```

2. Now, let's write a script to process a single user. We'll use `let` to grab the first user from the array for simplicity.

   `conditional_user.json`:

   ```
   ["let",
     [
       ["user", ["get", ["$input"], "/users/0"]]
     ],
     ["if",
       ["get", ["$", "/user"], "/active"],
       { "status": "Welcome!", "user_data": ["$", "/user"] },
       { "status": "Access Denied", "reason": "User is inactive" }
     ]
   ]
   ```

3. Run the script:

   ```
   computo conditional_user.json users_input.json
   ```

The output will be:

```
{
  "status": "Welcome!",
  "user_data": {
    "active": true,
    "name": "Alice",
    "plan": "premium"
  }
}
```

Because the `active` field for the first user (`Alice`) is `true`, the `then_expression` was evaluated and returned.

Now, change the `get` expression to select the second user, Bob: `["get", ["$input"], "/users/1"]`. Run the script again.

The output now reflects the `else_expression`:

```
{
  "reason": "User is inactive",
  "status": "Access Denied"
}
```

2.5.4 Pattern: Conditional Templating

The real power of `if` emerges when you combine it with `permuto.apply`. You can use Computo's logic to select the correct Permuto template based on input data.

Let's create two different Permuto templates.

`active_user_template.json`:

```
{
  "message": "Welcome back, ${/name}!",
  "dashboard_url": "/dashboard",
  "plan": "${/plan}"
}
```

inactive_user_template.json:

```
{
  "message": "Your account for ${/name} is inactive.",
  "reactivation_url": "/reactivate-account"
}
```

Now, we can write a script that uses `if` to choose which template to apply.

template_selector.json:

```
["let",
  [
    ["user", ["get", ["$input"], "/users/0"]]
  ],
  ["if",
    ["get", ["$", "/user"], "/active"],
    ["permuto.apply",
      {
        "message": "Welcome back, ${/name}!",
        "dashboard_url": "/dashboard",
        "plan": "${/plan}"
      },
      ["$", "/user"]
    ],
    ["permuto.apply",
      {
        "message": "Your account for ${/name} is inactive.",
        "reactivation_url": "/reactivate-account"
      },
      ["$", "/user"]
    ]
  ]
]
```

Run this script (with the `--interpolation` flag to handle the strings):

```
computo --interpolation template_selector.json users_input.json
```

Output for Alice (user 0):

```
{
  "dashboard_url": "/dashboard",
  "message": "Welcome back, Alice!",
  "plan": "premium"
}
```

Output for Bob (if you change the script to user 1):

```
{
  "message": "Your account for Bob is inactive.",
  "reactivation_url": "/reactivate-account"
}
```

This pattern is incredibly powerful. Your application logic remains clean; it simply executes a Computo script. The complex conditional templating logic is entirely self-contained within the script data itself.

2.5.5 In This Chapter

You've learned how to add decision-making to your transformations: * The syntax and behavior of the `if` operator. * The rules for **truthiness and falsiness** in Computo. * How to use `if` to return different JSON structures based on a condition. * The powerful pattern of using `if` to **conditionally apply different Permuto templates**.

So far we've only processed single data items. In the next chapter, we'll learn how to work with arrays, allowing us to process every item in a collection.

2.6 Chapter 6: Working with Arrays

In our previous examples, we manually selected a single user from an array with `["get", ..., "/users/0"]`. This is fine for demonstration, but real-world tasks require processing *every* item in a collection. You might need to transform a list of products, summarize a series of log entries, or, in our case, process a list of users.

This is where Computo's array operators come into play. We'll start with the most fundamental one: `map`.

2.6.1 The `map` Operator

The `map` operator iterates over an input array and applies a transformation to each item, producing a new array of the transformed items. The original array is not changed.

Its syntax introduces a new concept: the `lambda` (or anonymous function).

`["map", <array_expression>, ["lambda", ["<item_variable>"], <transform_expression>]]`

Let's break that down: * `<array_expression>`: An expression that must evaluate to a JSON array. * `["lambda", ... ]`: A special expression that defines an operation to be performed on each item. * `<item_variable>`: The name you choose for the variable that will hold the current item during each iteration. * `<transform_expression>`: The expression that transforms the item. Inside this expression, you can access the current item using `["$", "/<item_variable>"]`.

2.6.2 Your First `map`: Extracting Usernames

Let's use the same `users_input.json` from the last chapter. Our goal is to produce a simple JSON array containing only the names of all the users.

`users_input.json` (for reference):

```
{
  "users": [
    { "name": "Alice", "active": true, "plan": "premium" },
    { "name": "Bob", "active": false, "plan": "basic" },
    { "name": "Charlie", "active": true, "plan": "basic" }
  ]
}
```

1. Create a script named `get_names.json`.

   ```
   ["map",
     ["get", ["$input"], "/users"],
     ["lambda", ["user"], ["get", ["$", "/user"], "/name"]]
   ]
   ```

   Here is what this script tells the engine:

   a. "Map over the array found at the `/users` path in the input."
   b. "For each item, create a temporary variable named `user`."
   c. "As the transformation, `get` the `/name` property from the `user` variable."

2. Run the script:

```
computo get_names.json users_input.json
```

The output is a new array containing just the transformed items:

```
[
  "Alice",
  "Bob",
  "Charlie"
]
```

2.6.3 Unambiguous Array Syntax: `{"array": [...]}`

Before we continue, we need to address a crucial piece of syntax. In Computo, a JSON array `[...]` is always treated as an operator call. So how do you represent a *literal* array of data? You use a special object wrapper:

`{"array": [item1, item2, ...]}`

When the interpreter sees `{"array": ...}`, it knows you mean "this is a literal array of data," not an action to perform.

For example, to map over a hardcoded array, you would write:

```
["map",
  {"array": [10, 20, 30]},
  ["lambda", ["n"], ["+", ["$", "/n"], 5]]
]
```

This would produce `[15, 25, 35]`. This syntax prevents any ambiguity between operator calls and literal array data.

2.6.4 Transforming Objects within a `map`

The real power of `map` comes from transforming each item into a new object structure. You can use `obj` or `permuto.apply` right inside the `lambda`.

Let's transform our user list into a new list of simpler objects, each containing just a `name` and `plan`.

`transform_users.json`:

```
["map",
  ["get", ["$input"], "/users"],
  ["lambda", ["u"],
    ["obj",
      ["name", ["get", ["$", "/u"], "/name"]],
      ["plan", ["get", ["$", "/u"], "/plan"]]
    ]
  ]
]
```

Running this gives us a completely new array of objects:

```
[
  {
```

```
    "name": "Alice",
    "plan": "premium"
  },
  {
    "name": "Bob",
    "plan": "basic"
  },
  {
    "name": "Charlie",
    "plan": "basic"
  }
]
```

2.6.5 Combining `map` and `if`

You can use any Computo operator inside a `lambda`, including `if`. This allows for powerful, item-specific conditional logic.

Let's generate a status message for each user. Active users get a welcome message; inactive users get a notice.

`user_statuses.json`:

```
["map",
  ["get", ["$input"], "/users"],
  ["lambda", ["user"],
    ["if",
      ["get", ["$", "/user"], "/active"],
      ["permuto.apply",
        {"message": "User ${/name} is active."},
        ["$", "/user"]
      ],
      ["permuto.apply",
        {"message": "User ${/name} is INACTIVE."},
        ["$", "/user"]
      ]
    ]
  ]
]
```

Run this with the `--interpolation` flag:

```
computo --interpolation user_statuses.json users_input.json
```

The output is an array of conditionally generated objects:

```
[
  {
    "message": "User Alice is active."
  },
  {
```

```
    "message": "User Bob is INACTIVE."
  },
  {
    "message": "User Charlie is active."
  }
]
```

2.6.6 In This Chapter

You've added array processing to your skillset. You have learned: * How to iterate over an array and transform each item using the **`map`** operator. * The syntax for **`lambda`** expressions to define the per-item transformation. * The special **`{"array": [...]}`** syntax for representing literal arrays. * How to combine `map` with `obj`, `if`, and `permuto.apply` for complex list transformations.

`map` is the first of several array operators. In the next chapters, we will explore others like `filter` and `reduce` to further refine our data pipelines.

2.7 Chapter 7: Object Construction and Manipulation

In modern data exchange, much of the work involves creating, reshaping, and merging JSON objects. In previous chapters, we've seen how to construct objects piece-by-piece using `obj`. This chapter will formalize those patterns and introduce new tools for more complex object manipulation.

2.7.1 Review: The `obj` Operator

As a quick refresher, the `obj` operator is the most fundamental way to build a new JSON object. It takes a series of `[key, value]` pairs and assembles them into an object.

```
["obj", ["key1", <value_expr1>], ["key2", <value_expr2>], ...]
```

The keys must be literal strings, but the values can be any valid Computo expression. This allows you to construct objects from a mix of static and dynamic data.

Example: Building a metadata object

```
["obj",
  ["timestamp", 1672531200],
  ["source", "api-gateway"],
  ["user_id", ["get", ["$input"], "/user/id"]],
  ["is_premium_user", [">", ["get", ["$input"], "/user/credits"], 100]]
]
```

This expression creates a new object by: * Using two literal values (`1672531200` and `"api-gateway"`). * Extracting a value directly from the input using `get`. * Calculating a boolean value using the `>` comparison operator.

2.7.2 New Operator: `merge`

A common task is to combine multiple objects. For example, you might have a set of default configuration values that you want to override with user-specific settings. Doing this manually with `obj` and `if` would be tedious.

Computo provides the `merge` operator for this exact purpose.

```
["merge", <object1>, <object2>, <object3>, ...]
```

The `merge` operator takes two or more expressions that evaluate to objects and combines them into a single new object. If multiple source objects contain the same key, the value from the **rightmost** object wins.

2.7.2.1 `merge` Example: Default and User Settings

Let's imagine our application has default settings, but we want to allow a user to override them.

1. **`settings_input.json`:** `json { "user_preferences": { "theme": "dark", "notifications": { "email": false } } }`
2. **`merge_settings.json`:** In this script, we'll define a default settings object and merge the user's preferences over it.

```
["let",
  [
    ["defaults", {"obj":
      [
        ["theme", "light"],
        ["notifications", {"obj": [
          ["email", true],
          ["sms", true]
        ]}],
        ["language", "en-US"]
      ]}
    ],
    ["user_prefs", ["get", ["$input"], "/user_preferences"]]
  ],
  ["merge", ["$", "/defaults"], ["$", "/user_prefs"]]
]
```

3. Run the script:

   ```
   computo merge_settings.json settings_input.json
   ```

Output:

```
{
  "language": "en-US",
  "notifications": {
    "email": false
  },
  "theme": "dark"
}
```

Let's analyze the result: * `language`: `"en-US"` was preserved from the default settings because it didn't exist in the user's preferences. * `theme`: `"dark"` from the user's preferences (the rightmost object) overwrote the default of `"light"`. * `notifications`: This is a **shallow merge**. The `notifications` *object* from the user preferences completely replaced the default `notifications` object. The `sms: true` key from the defaults is gone.

This shallow merge behavior is intentional and predictable. For a "deep" or recursive merge, you would need to apply `merge` at each level of the object hierarchy.

2.7.3 Pattern: Creating Dynamic Keys

A limitation of the `obj` operator is that keys must be literal strings. You cannot use an expression to generate a key name dynamically.

While Computo doesn't have a direct "dynamic key" operator, you can achieve this result by combining `map` and `merge`. The pattern is to create an array of single-key objects and then merge them all together.

Let's say we have an array of settings and we want to turn it into a key-value map.

1. **`kv_input.json`:** `json { "settings_list": [ { "key": "user_theme",`

```
"value": "dark" },         { "key": "user_font_size", "value": 14 },         {
"key": "user_id", "value": "u-456" }       ]     }
```

2. **`dynamic_keys.json`:** This script is a bit more advanced. It uses `map` to turn each item into a single-key object, and then uses `reduce` (which we'll cover in detail later) with `merge` to combine them.

```
["reduce",
  ["map",
    ["get", ["$input"], "/settings_list"],
    ["lambda", ["item"],
      ["obj",
        [
          ["get", ["$", "/item"], "/key"],
          ["get", ["$", "/item"], "/value"]
        ]
      ]
    ]
  ],
  ["lambda", ["acc", "obj"], ["merge", ["$", "/acc"], ["$", "/obj"]]],
  {}
]
```

Note: This is an advanced preview of ***`reduce`****. Don't worry if it's not fully clear yet; the key takeaway is the* ***`map`****-then-*`merge` *pattern.*

Output:

```
{
  "user_font_size": 14,
  "user_id": "u-456",
  "user_theme": "dark"
}
```

This pattern demonstrates how you can compose Computo's core operators to achieve sophisticated results that might otherwise seem to require a dedicated operator.

2.7.4 In This Chapter

You've deepened your understanding of how to work with objects in Computo. * You've reviewed the **`obj`** operator for explicit object construction. * You've learned to use the **`merge`** operator to combine multiple objects, with a clear "right-most wins" rule for conflicting keys. * You've seen an advanced pattern for creating objects with **dynamic keys** by composing `map` and `merge`.

With these tools, you have complete control over the shape and content of the JSON objects you produce. In the next chapter, we will return to arrays and explore the remaining powerhouse operators: `filter` and `reduce`.

2.8 Chapter 8: Advanced Array Operations

In Chapter 6, we introduced `map`, the fundamental tool for transforming arrays. However, transformation is only one part of the story. Often, you need to select a subset of items from an array or aggregate an entire array into a single value.

This chapter introduces the remaining core array operators: `filter` and `reduce`. Mastering these three operators (`map`, `filter`, and `reduce`) will give you a complete and powerful toolkit for virtually any array manipulation task.

2.8.1 `filter`: Selecting Items from an Array

While `map` transforms every item in an array, `filter` *selects* items from an array. It iterates over an array and returns a new array containing only the items for which a given condition is "truthy".

The syntax is nearly identical to `map`:

`["filter", <array_expression>, ["lambda", ["<item_variable>"], <condition_expression>]]`

The `<condition_expression>` must evaluate to a "truthy" or "falsy" value, following the same rules as the `if` operator. If the condition is truthy, the item is kept; if falsy, it is discarded.

2.8.1.1 `filter` Example: Finding Active Users

Let's return to our `users_input.json` and create a new list containing only the active users.

`users_input.json` (for reference):

```
{
  "users": [
    { "name": "Alice", "active": true, "plan": "premium" },
    { "name": "Bob", "active": false, "plan": "basic" },
    { "name": "Charlie", "active": true, "plan": "basic" }
  ]
}
```

`filter_active.json`:

```
["filter",
  ["get", ["$input"], "/users"],
  ["lambda", ["user"], ["get", ["$", "/user"], "/active"]]
]
```

The lambda here is simple: for each `user` object, it gets the value of the `active` key. Since `true` is truthy and `false` is falsy, this works perfectly as a condition.

Output:

```
[
  {
    "active": true,
    "name": "Alice",
    "plan": "premium"
  },
  {
```

```
    "active": true,
    "name": "Charlie",
    "plan": "basic"
  }
]
```

The resulting array contains only the objects for Alice and Charlie, because Bob's `active` field was `false`.

2.8.2 Chaining `filter` and `map`

The true power of these operators comes from chaining them together. Because `filter` returns an array, you can feed its result directly into a `map` operator.

Let's build on the previous example. We want a list of just the *names* of the active, premium users.

`active_premium_names.json`:

```
["map",
  ["filter",
    ["filter",
      ["get", ["$input"], "/users"],
      ["lambda", ["user"], ["get", ["$", "/user"], "/active"]]
    ],
    ["lambda", ["user"], ["==", ["get", ["$", "/user"], "/plan"], "premium"]]
  ],
  ["lambda", ["user"], ["get", ["$", "/user"], "/name"]]
]
```

This script looks complex, but it's a very clear data pipeline if you read it from the inside out: 1. **First `filter`**: Selects all users where `active` is true. 2. **Second `filter`**: Takes the result of the first filter and, from that subset, selects all users where `plan` is equal to `"premium"`. 3. **`map`**: Takes the final filtered list (which now only contains Alice) and transforms it into an array of names.

Output:

```
[
  "Alice"
]
```

This pattern—**filter, then map**—is one of the most common and powerful patterns in functional data processing.

2.8.3 `reduce`: Aggregating an Array to a Single Value

While `map` and `filter` produce new arrays, `reduce` (sometimes called "fold" or "accumulate") boils an entire array down to a single value. This is used for tasks like summing numbers, concatenating strings, or flattening a list of lists.

The `reduce` operator is the most complex of the three, introducing an "accumulator".

`["reduce", <array_expression>, <lambda>, <initial_value>]`

The `lambda` for `reduce` takes **two** arguments: `["lambda", ["<accumulator>", "<current_item>"], <expression>]`

Here's how it works: 1. The `<accumulator>` is initialized with the `<initial_value>`. 2. The lambda is called for the first item in the array. The result of its `<expression>` becomes the **new value of the accumulator**. 3. The lambda is called for the second item, using the updated accumulator. This repeats for all items. 4. The final value of the accumulator is the result of the `reduce` operation.

2.8.3.1 `reduce` Example: Summing an Array

Let's calculate the total cost of all user plans in our system.

`total_cost.json`:

```
["let",
  [
    ["costs", {"array": [20, 5, 5]}]
  ],
  ["reduce",
    ["$", "/costs"],
    ["lambda", ["total", "cost"], ["+", ["$", "/total"], ["$", "/cost"]]],
    0
  ]
]
```

1. `total` starts at `0` (the initial value).
2. **Iteration 1:** `total` is `0`, `cost` is `20`. The lambda returns `0 + 20 = 20`. `total` is now `20`.
3. **Iteration 2:** `total` is `20`, `cost` is `5`. The lambda returns `20 + 5 = 25`. `total` is now `25`.
4. **Iteration 3:** `total` is `25`, `cost` is `5`. The lambda returns `25 + 5 = 30`. `total` is now `30`.
5. The array is exhausted. `reduce` returns the final value of the accumulator.

Output: `30`

2.8.4 The Complete Pipeline: Map, Filter, Reduce

Now we can combine all three to answer complex questions. For example: "What is the total monthly cost of all active, premium plans?"

`premium_revenue.json`:

```
["reduce",
  ["map",
    ["filter",
      ["get", ["$input"], "/users"],
      ["lambda", ["u"],
        ["&&",
          ["get", ["$", "/u"], "/active"],
          ["==", ["get", ["$", "/u"], "/plan"], "premium"]
        ]
      ]
    ],
```

```
    ["lambda", ["u"], ["get", ["$", "/u"], "/monthly_cost"]]
  ],
  ["lambda", ["total", "cost"], ["+", ["$", "/total"], ["$", "/cost"]]],
  0
]
```

(Note: This example uses a logical AND operator, `&&`, which we haven't formally covered but whose function is intuitive here. We'll cover it later.)

The Pipeline: 1. **`filter`**: Finds all users who are both active AND have a premium plan. (Result: just Alice's object). 2. **`map`**: Takes that filtered list and transforms it into a list of costs. (Result: `[20]`). 3. **`reduce`**: Takes that list of costs and sums it. (Result: `20`).

Output: `20`

2.8.5 In This Chapter

You have now completed the core functional toolkit for array processing. * You learned to use **`filter`** to selectively create new arrays based on a condition. * You learned to use **`reduce`** to aggregate an array's contents into a single result. * You saw how **chaining `filter` and `map`** creates powerful data pipelines. * You combined all three operators—**`filter`, `map`, and `reduce`**—to answer a complex question about a data set in a single, expressive script.

You are now equipped to handle a vast range of data transformation challenges. The following chapters will build on this foundation, exploring how to apply these patterns to real-world integration scenarios.

2.9 Chapter 9: Template-Driven Transformations

Welcome to Part III of our guide. By now, you have a solid grasp of Computo's logic and Permuto's templating. You can make decisions with `if`, iterate with `map`, and reshape data with `permuto.apply`. Now, it's time to elevate our approach and learn the patterns that separate simple scripts from robust, maintainable data pipelines.

This chapter focuses on a powerful strategy: **Template-Driven Transformation**. The core idea is to let a clean, declarative Permuto template define the "what" (the final data structure), while a focused Computo script handles the "how" (the logic needed to prepare the data for that template).

2.9.1 The Core Pattern: Dynamic Context Building

So far, we have often passed our raw input directly to `permuto.apply` like this:

```
["permuto.apply", <template>, ["$input"]]
```

This works, but it tightly couples your template to the (potentially messy) structure of your input data. If the input API changes a field name, you have to change your template.

A more robust pattern is to use Computo to **build a clean, intermediate context object first**. This new context is tailored specifically to what the template expects.

The Workflow: 1. Define a "perfect" output structure in a Permuto template. 2. Write a Computo script that reads the messy raw input. 3. The script uses `map`, `filter`, `let`, and `obj` to build a new, clean context object. 4. The script's final step is to call `permuto.apply`, passing it the template and the **newly-built context**.

This decouples your final output from your raw input, making your transformations more resilient to change.

2.9.1.1 Example: Generating a Product Display Card

Let's imagine we're receiving a product data object from a backend service, and we need to transform it into a format suitable for a UI display card.

1. **The Raw Input (`product_input.json`):** This data is a bit messy. It has internal IDs, a list of variants with stock counts, and prices in cents. `json { "product_id": "prod-xyz-789", "base_price_cents": 2999, "name": "Quantum T-Shirt", "variants": [{ "sku": "xyz-s-red", "attrs": { "size": "S", "color": "Red" }, "stock": 10 }, { "sku": "xyz-m-red", "attrs": { "size": "M", "color": "Red" }, "stock": 0 }, { "sku": "xyz-l-blue", "attrs": { "size": "L", "color": "Blue" }, "stock": 25 }], "metadata": { "source": "warehouse-api" } }`

2. **The Target Template:** Our UI card needs a clean, simple structure. We'll define this in a Permuto template. Notice this template expects a `price` (in dollars), a `title`, and a list of `available_options`. It knows nothing about `base_price_cents` or `stock`.

   `card_template.json`:

   ```
   {
     "display_card": {
       "title": "${/title}",
   ```

```
    "price": "USD $${/price}",
    "options": "${/available_options}"
  }
}
```

3. **The Computo "Glue" Script (`build_card_context.json`):** This is where the magic happens. This script will read the raw input and build the perfect context for `card_template.json`.

```
["let",
  [
    ["product", ["$input"]],
    ["in_stock_variants",
      ["filter",
        ["get", ["$", "/product"], "/variants"],
        ["lambda", ["v"], [">", ["get", ["$", "/v"], "/stock"], 0]]
      ]
    ]
  ],
  ["let",
    [
      ["clean_context",
        ["obj",
          ["title", ["get", ["$", "/product"], "/name"]],
          ["price", ["/", ["get", ["$", "/product"], "/base_price_cents"], 100]],
          ["available_options",
            ["map",
              ["$", "/in_stock_variants"],
              ["lambda", ["v"], ["get", ["$", "/v"], "/attrs"]]
            ]
          ]
        ]
      ]
    ],
    ["permuto.apply",
      {
        "display_card": {
          "title": "${/title}",
          "price": "USD $${/price}",
          "options": "${/available_options}"
        }
      },
      ["$", "/clean_context"]
    ]
  ]
]
```

Let's analyze the script's logic: * It uses `let` to store the raw `product` data. * It uses `filter` to create a new array, `in_stock_variants`, containing only variants with `stock > 0`. * It enters

a new `let` block to build the `clean_context`. * The `title` is extracted directly. * The `price` is *calculated* by dividing `base_price_cents` by 100. * `available_options` is created by *mapping* over the `in_stock_variants` and extracting just the `attrs` object from each. * Finally, it calls `permuto.apply`, passing our hardcoded template and the newly created `clean_context` variable.

Run it with the `--interpolation` flag for the price string:

```
computo --interpolation build_card_context.json product_input.json
```

Final Output:

```
{
  "display_card": {
    "options": [
      {
        "color": "Red",
        "size": "S"
      },
      {
        "color": "Blue",
        "size": "L"
      }
    ],
    "price": "USD $29.99",
    "title": "Quantum T-Shirt"
  }
}
```

We successfully transformed a messy backend object into a perfect UI-ready object by separating the data preparation logic from the final presentation template.

2.9.2 In This Chapter

This chapter shifted our focus from learning operators to learning strategy. * You learned the **Dynamic Context Building** pattern, a robust method for decoupling templates from raw data structures. * You saw how to use a chain of Computo operators (`let`, `filter`, `map`, `obj`) to prepare a clean dataset. * You reinforced the clear division of labor: **Computo prepares the data, Permuto presents the data.**

This pattern is the key to creating scalable and maintainable transformations. In the next chapter, we'll look at other common data pipeline patterns that build on this foundation.

2.10 Chapter 10: Data Pipeline Patterns

In the previous chapter, we introduced the most important transformation strategy: building a dynamic context before applying a template. This chapter expands on that idea by introducing several other common and powerful data pipeline patterns.

Think of these as reusable blueprints for your transformations. Recognizing these patterns will help you structure your scripts more effectively and solve complex problems with surprising clarity.

2.10.1 Pattern 1: The Enrichment Pipeline

Goal: To take an existing data structure and add new, computed information to it.

When to use: When your input data is mostly correct, but you need to augment it with additional fields, flags, or calculations before passing it on.

The key operator for this pattern is **`merge`**. You start with the original object and merge a new object containing your computed fields onto it.

2.10.1.1 Enrichment Example: Scoring User Engagement

Imagine we have a user object and we want to add an `engagement_score` and a `status` flag based on their activity.

1. **Input Data (`user_activity.json`):** `json { "user_id": "u-777", "logins": 42, "posts_created": 15, "comments_made": 112, "last_login_day 3 }`

2. **The Script (`enrich_user.json`):** This script calculates a score and determines a status, then merges this new information back into the original user object.

```
["let",
  [
    ["user", ["$input"]],
    ["score",
      ["+",
        ["*", ["get", ["$", "/user"], "/logins"], 2],
        ["*", ["get", ["$", "/user"], "/posts_created"], 10],
        ["*", ["get", ["$", "/user"], "/comments_made"], 5]
      ]
    ]
  ],
  ["merge",
    ["$", "/user"],
    ["obj",
      ["engagement_score", ["$", "/score"]],
      ["status",
        ["if",
          ["&&",
            [">", ["$", "/score"], 500],
            ["<", ["get", ["$", "/user"], "/last_login_days_ago"], 30]
          ],
          "active_and_high_engagement",
```

```
          "at_risk"
        ]
      ]
    ]
  ]
]
```

3. **The Output:** The result is the original object, plus the two new fields. `json { "comments_made": 112, "engagement_score": 794, "last_login_days_ago": 3, "logins": 42, "posts_created": 15, "status": "active_and_high_engagem "user_id": "u-777" }` This pattern is perfect for middleware, where you receive data, add value to it, and then pass it to the next service in the chain.

2.10.2 Pattern 2: The Forking Pipeline

Goal: To create two or more completely different output structures from a single input source.

When to use: When you need to send tailored data to different downstream systems. For example, one payload for your analytics service and another for your search index.

This pattern uses a top-level `obj` or `permuto.apply` to define the separate output "forks," with each fork containing its own independent transformation logic.

2.10.2.1 Forking Example: Indexing and Analytics From a single raw event, we need to generate a compact document for our search index and a detailed document for our analytics database.

1. **Input Data (`raw_event.json`):** `json { "event_id": "evt-aaa-bbb", "timestamp": "2023-10-27T10:00:00Z", "user": { "id": "u-123", "name": "Alice" }, "action": "product_view", "details": { "product_id": "prod-xyz-789", "product_name": "Quantum T-Shirt", "price_cents": 2999, "tags": ["apparel", "t-shirt", "quantum"] } }`

2. **The Script (`forking_pipeline.json`):** The top-level expression is an `obj` with two keys, `for_search` and `for_analytics`. Each value is a `let` block that defines a self-contained transformation.

```
["obj",
  ["for_search",
    ["let", [["d", ["get", ["$input"], "/details"]]],
      ["obj",
        ["doc_id", ["get", ["$", "/d"], "/product_id"]],
        ["content", ["get", ["$", "/d"], "/product_name"]],
        ["tags", ["get", ["$", "/d"], "/tags"]]
      ]
    ]
  ],
  ["for_analytics",
    ["let", [["d", ["get", ["$input"], "/details"]]],
      ["obj",
        ["event_type", ["get", ["$input"], "/action"]],
```

```
        ["user", ["get", ["$input"], "/user/id"]],
        ["product", ["get", ["$", "/d"], "/product_id"]],
        ["revenue_potential", ["/", ["get", ["$", "/d"], "/price_cents"], 100]]
      ]
    ]
  ]
]
```

3. **The Output:** The final JSON contains two distinct, independent objects, ready to be sent to their respective systems. `json { "for_analytics": { "event_type": "product_view", "product": "prod-xyz-789", "revenue_potentia 29.99, "user": "u-123" }, "for_search": { "content": "Quantum T-Shirt", "doc_id": "prod-xyz-789", "tags": [ "apparel", "t-shirt", "quantum" ] } }`

2.10.3 Pattern 3: The Aggregation Pipeline

Goal: To process a list of items and produce a single summary object.

When to use: When you need to generate reports, dashboards, or summary statistics from a collection of raw data.

This pattern typically involves heavy use of `filter`, `map`, and `reduce` to calculate summary statistics, which are then assembled into a final object using `obj`.

2.10.3.1 Aggregation Example: Sales Report

Let's generate a summary report from a list of sales transactions.

1. **Input Data (`sales_data.json`):** `json { "transactions": [{ "id": 1, "product": "A", "amount": 100, "region": "NA" }, { "id": 2, "product": "B", "amount": 150, "region": "EU" }, { "id": 3, "product": "A", "amount": 120, "region": "NA" }, { "id": 4, "product": "C", "amount": 200, "region": "NA" }, { "id": 5, "product": "B", "amount": 180, "region": "APAC" }] }`

2. **The Script (`sales_report.json`):** This script uses multiple `let` bindings to calculate different statistics before assembling the final report.

```
["let",
  [
    ["txs", ["get", ["$input"], "/transactions"]],
    ["all_amounts", ["map", ["$", "/txs"], ["lambda",["t"],["get",["$","/t"],"/amount"]]]]
    ["na_txs", ["filter", ["$", "/txs"], ["lambda",["t"],["==",["get",["$","/t"],"/region"
  ],
  ["obj",
    ["total_transactions", ["count", ["$", "/txs"]]],
    ["total_revenue",
      ["reduce", ["$", "/all_amounts"], ["lambda",["acc","n"],["+",["$","/acc"],["$","/n"]
    ],
    ["avg_revenue",
      ["/",
```

```
        ["reduce", ["$", "/all_amounts"], ["lambda",["acc","n"],["+",["$","/acc"],["$","/n
        ["count", ["$", "/txs"]]
      ]
    ],
    ["na_revenue",
      ["reduce",
        ["map", ["$", "/na_txs"], ["lambda",["t"],["get",["$","/t"],"/amount"]]],
        ["lambda",["acc","n"],["+",["$","/acc"],["$","/n"]]], 0
      ]
    ]
  ]
]
```

Note: `count` is a simple new operator that returns the length of an array.

3. **The Output:** The script produces a single JSON object summarizing the raw transaction data. `json { "avg_revenue": 150, "na_revenue": 420, "total_revenue": 750, "total_transactions": 5 }`

2.10.4 In This Chapter

You've learned three fundamental blueprints for structuring your transformations: * **Enrichment:** Adding computed data to an existing object using `merge`. * **Forking:** Creating multiple, distinct outputs from one source. * **Aggregation:** Summarizing a list of items into a single report object using `map`, `filter`, and `reduce`.

By recognizing these patterns in your own data challenges, you can write cleaner, more intuitive, and more powerful Computo scripts.

2.11 Chapter 11: Complex Real-World Examples

You have mastered the operators and learned the fundamental patterns. Now it is time to put it all together. This chapter walks through a single, complex, real-world problem from start to finish. We will not introduce any new operators. Instead, we will focus on how the tools you already know can be composed to build a sophisticated, multi-stage data transformation pipeline.

2.11.1 The Scenario: E-commerce Order Processing

Imagine we are building the backend for an e-commerce platform. When a customer places an order, our system receives a single, rich JSON object representing that order. Our task is to process this object and generate **three distinct outputs** for different downstream systems:

1. **A Customer Invoice:** A simplified object containing the customer's details, the items purchased with calculated subtotals, and a final grand total including tax.
2. **A Shipping Manifest:** A document for the warehouse containing the customer's shipping address and a list of only the *physical* items to be packed, including their SKUs and quantities.
3. **An Inventory Update:** A list of operations for the inventory system, detailing which SKUs need their stock levels decremented.

This is a perfect example of a **Forking Pipeline**, where each fork (`invoice`, `shipping`, `inventory`) will require its own unique transformation logic.

2.11.2 The Input Data

Here is the raw order data we'll be working with. Note that it contains a mix of physical and digital goods.

order_input.json:

```
{
  "order_id": "ord-112233",
  "customer": {
    "user_id": "u-456",
    "name": "Jane Doe",
    "email": "jane.doe@example.com",
    "shipping_address": {
      "street": "123 Quantum Lane",
      "city": "Photon Creek",
      "zip": "98765",
      "country": "USA"
    }
  },
  "line_items": [
    { "item_id": "prod-xyz-789", "sku": "QC-TS-M-BLUE", "name": "Quantum T-Shirt", "quantity":
    { "item_id": "prod-abc-456", "sku": "QC-MUG-BLACK", "name": "Flux-Capacitor Mug", "quantity
    { "item_id": "dig-001", "sku": "EBOOK-COMPUTO", "name": "Computo: The Ultimate Guide", "qua
  ],
  "tax_rate_percent": 8
}
```

2.11.3 The Plan of Attack

Our top-level output will be a single JSON object with three keys: `invoice`, `shipping_manifest`, and `inventory_updates`.

1. **`invoice`:** We will need to map over the `line_items`, calculate a `subtotal_cents` for each, and then `reduce` them to calculate a `grand_total_cents` after applying tax.
2. **`shipping_manifest`:** We must first `filter` the `line_items` to get only those where `physical` is `true`. Then, we'll `map` that filtered list into the simple format the warehouse needs.
3. **`inventory_updates`:** This will be similar to the shipping manifest, but it will produce a list of decrement operations.

2.11.4 The Complete Transformation Script

This script is the most complex we've written, but by using `let` to break down the problem, it remains readable.

`process_order.json`:

```
["let",
  [
    ["order", ["$input"]],
    ["items", ["get", ["$", "/order"], "/line_items"]],
    ["physical_items", ["filter", ["$", "/items"], ["lambda",["i"],["get",["$","/i"],"/physica
  ],
  ["obj",
    ["invoice",
      ["let",
        [
          ["items_with_subtotals",
            ["map", ["$", "/items"],
              ["lambda", ["item"],
                ["merge", ["$", "/item"],
                  ["obj", ["subtotal_cents", ["*",["get",["$","/item"],"/quantity"],["get",["$
                ]
              ]
            ]
          ],
          ["pre_tax_total", ["reduce", ["$", "/items_with_subtotals"], ["lambda",["acc","i"],["
        ],
        ["obj",
          ["customer_name", ["get", ["$", "/order"], "/customer/name"]],
          ["items", ["$", "/items_with_subtotals"]],
          ["grand_total_cents",
            ["+",
              ["$", "/pre_tax_total"],
              ["*", ["$", "/pre_tax_total"], ["/", ["get", ["$", "/order"], "/tax_rate_percent
            ]
          ]
        ]
```

```
      ]
    ],
    ["shipping_manifest",
      ["obj",
        ["order_id", ["get", ["$", "/order"], "/order_id"]],
        ["shipping_address", ["get", ["$", "/order"], "/customer/shipping_address"]],
        ["items_to_pack",
          ["map", ["$", "/physical_items"],
            ["lambda", ["item"],
              ["obj",
                ["sku", ["get", ["$", "/item"], "/sku"]],
                ["quantity", ["get", ["$", "/item"], "/quantity"]],
                ["description", ["get", ["$", "/item"], "/name"]]
              ]
            ]
          ]
        ]
      ]
    ],
    ["inventory_updates",
      ["map", ["$", "/physical_items"],
        ["lambda", ["item"],
          ["obj",
            ["action", "DECREMENT_STOCK"],
            ["sku", ["get", ["$", "/item"], "/sku"]],
            ["by_quantity", ["get", ["$", "/item"], "/quantity"]]
          ]
        ]
      ]
    ]
  ]
]
```

2.11.5 The Final Output

Running the script produces a single, comprehensive JSON object with all three required outputs, each perfectly structured for its destination.


```
{
  "inventory_updates": [
    { "action": "DECREMENT_STOCK", "by_quantity": 1, "sku": "QC-TS-M-BLUE" },
    { "action": "DECREMENT_STOCK", "by_quantity": 2, "sku": "QC-MUG-BLACK" }
  ],
  "invoice": {
    "customer_name": "Jane Doe",
    "grand_total_cents": 7666.92,
    "items": [
      { "item_id": "prod-xyz-789", "name": "Quantum T-Shirt", "physical": true, "price_cents":
```

```
      { "item_id": "prod-abc-456", "name": "Flux-Capacitor Mug", "physical": true, "price_cents
      { "item_id": "dig-001", "name": "Computo: The Ultimate Guide", "physical": false, "price_
    ]
  },
  "shipping_manifest": {
    "items_to_pack": [
      { "description": "Quantum T-Shirt", "quantity": 1, "sku": "QC-TS-M-BLUE" },
      { "description": "Flux-Capacitor Mug", "quantity": 2, "sku": "QC-MUG-BLACK" }
    ],
    "order_id": "ord-112233",
    "shipping_address": { "city": "Photon Creek", "country": "USA", "street": "123 Quantum Lane
  }
}
```

2.11.6 In This Chapter

This example was a microcosm of everything you've learned. * We used a top-level **forking pipeline** to create multiple outputs. * The `invoice` fork used the **enrichment** (`merge`) and **aggregation** (`reduce`) patterns. * The `shipping_manifest` fork used a **filter-then-map** pipeline to process a subset of data. * We used `let` extensively to break the problem into logical, readable steps, proving that even complex scripts can be managed effectively.

You are now fully equipped to design and implement sophisticated JSON transformations to solve real-world integration challenges. The final part of this guide will cover advanced topics like performance, error handling, and best practices.

2.12 Chapter 12: Performance and Optimization

You've now seen how to solve complex problems with Computo and Permuto. As you move from writing small scripts to building business-critical data pipelines, performance becomes an important consideration.

The Computo engine is designed to be efficient, but the structure of your script can have a significant impact on its execution speed and memory usage. This chapter provides guidance on writing high-performance transformations and understanding the trade-offs involved.

2.12.1 The Golden Rule: `let` is Your Best Friend

The single most important optimization technique in Computo is the proper use of the `let` operator.

Anti-Pattern: Re-evaluating Expressions Consider this script that uses the same `get` expression multiple times:

```
["obj",
  ["name", ["get", ["get", ["$input"], "/user"], "/profile"], "/name"]],
  ["email", ["get", ["get", ["$input"], "/user"], "/profile"], "/email"]]
]
```

The expression `["get", ["$input"], "/user"]` is evaluated twice. While this is a small example, in a complex script with many nested operators, this redundant work can add up.

Optimized Pattern: Bind Once, Use Many Times By binding the result of an expensive or frequently used expression to a variable, you ensure it is evaluated only once.

```
["let",
  [
    ["user_profile", ["get", ["get", ["$input"], "/user"], "/profile"]]
  ],
  ["obj",
    ["name", ["get", ["$", "/user_profile"], "/name"]],
    ["email", ["get", ["$", "/user_profile"], "/email"]]
  ]
]
```

This version is not only faster and more memory-efficient, but it's also significantly more readable. **When in doubt, use `let` to store the result of any non-trivial expression that you plan to use more than once.**

2.12.2 Understanding Lazy Evaluation

Computo's `if` operator is "lazy." This means it only evaluates the branch that is actually chosen. The other branch is never touched. This has important performance implications.

Inefficient: Evaluating Before the `if`

```
["let",
  [
    ["premium_dashboard", <... very expensive expression to build a dashboard ...>],
    ["basic_dashboard", <... another expensive expression ...>]
```

```
  ],
  ["if",
    ["get", ["$input"], "/user/is_premium"],
    ["$", "/premium_dashboard"],
    ["$", "/basic_dashboard"]
  ]
]
```

In this script, **both** the premium and basic dashboards are fully computed and stored in variables, even though only one will ever be used.

Efficient: Evaluating Inside the `if` By moving the expensive expressions inside the `if` branches, you ensure that only the necessary work is done.

```
["if",
  ["get", ["$input"], "/user/is_premium"],
  <... very expensive expression to build a dashboard ...>,
  <... another expensive expression ...>
]
```

This is a critical pattern for performance. **Defer expensive computations by placing them inside the branches of an `if` statement whenever possible.**

2.12.3 Operator Performance Characteristics

Not all operators are created equal. Here is a general guide to their relative performance cost:

- **Low Cost (Fast):** `+`, `-`, `*`, `/`, `==`, `>`, `get`, `$`, `obj`. These are typically very fast, often mapping to single machine instructions or efficient hash map lookups.
- **Medium Cost (Depends on Data Size):** `merge`, `permuto.apply`. The cost of these operators is proportional to the size of the objects or templates they are processing.
- **High Cost (Iterative):** `map`, `filter`, `reduce`. These are the most powerful operators, but also the most expensive. Their cost is directly proportional to the number of items in the array they are iterating over. A `map` over an array with 1,000,000 items will be 1,000 times slower than a `map` over an array with 1,000 items.

2.12.4 Pipeline Ordering Matters

The order of your chained array operations can have a massive impact on performance. The key principle is to **reduce the size of your dataset as early as possible.**

Anti-Pattern: `map` before `filter` Imagine you need to get the names of all active users.

```
["map",
  ["filter",
    ["map",
      ["get", ["$input"], "/users"],
      ["lambda", ["u"], ["obj", ["name", ["get",...]], ["active", ["get",...]]]]
    ],
    ["lambda", ["u"], ["get", ["$", "/u"], "/active"]]
  ],
```

```
  ["lambda", ["u"], ["get", ["$", "/u"], "/name"]]
]
```

This is highly inefficient. If you have 10,000 users, the first `map` operation will create 10,000 new, temporary objects in memory. Then, `filter` will iterate over those 10,000 new objects and likely discard most of them.

Optimized Pattern: `filter` before `map`

```
["map",
  ["filter",
    ["get", ["$input"], "/users"],
    ["lambda", ["u"], ["get", ["$", "/u"], "/active"]]
  ],
  ["lambda", ["u"], ["get", ["$", "/u"], "/name"]]
]
```

This version is dramatically better. The `filter` operator runs first on the raw input. If only 100 users are active, the subsequent `map` only has to do work on those 100 items. It creates far fewer temporary objects and performs fewer iterations.

Always filter as early as you can to reduce the amount of data that later stages in your pipeline need to process.

2.12.5 Tail Call Optimization: No Stack Overflow Worries

Computo implements tail call optimization (TCO), which means you don't need to worry about stack overflow errors from deeply nested control flow structures. This makes Computo suitable for complex, programmatically-generated transformations.

2.12.5.1 What This Means for You

Traditional recursive programs can cause stack overflow if they nest too deeply. Computo eliminates this concern:

```
// This would cause stack overflow in many languages, but not in Computo
["if", ["count", ["$input"]],
  ["if", [">", ["get", ["$input"], "/score"], 90],
    ["if", ["==", ["get", ["$input"], "/status"], "active"],
      ["if", ["!=", ["get", ["$input"], "/role"], "admin"],
        "Process user",
        "Skip admin"
      ],
      "Inactive user"
    ],
    "Low score"
  ],
  "Empty input"
]
```

2.12.5.2 Practical Benefits

1. **Complex conditional trees**: You can nest `if` statements arbitrarily deep without performance degradation
2. **Deep `let` scoping**: Nested variable scopes don't consume stack space
3. **Programmatic generation**: Generated scripts with deep nesting work reliably
4. **No configuration needed**: TCO is automatic and always enabled

This optimization makes Computo robust for complex business logic and machine-generated transformations where nesting depth might be unpredictable.

2.12.6 In This Chapter

You've learned the core principles of writing high-performance Computo scripts: * Use **`let`** to avoid re-evaluating expressions. * Leverage the **lazy evaluation** of the `if` operator to defer expensive work. * Understand the relative performance costs of different operators. * Structure your array pipelines to **`filter` early and `map` late**. * Take advantage of **tail call optimization** for deeply nested control flow without stack overflow concerns.

By applying these principles, you can ensure that your transformations are not only correct but also fast and efficient enough for production workloads. The final chapters will cover error handling and best practices to round out your expertise.

2.13 Chapter 13: Error Handling and Debugging

In an ideal world, all of our scripts would run perfectly on the first try. In the real world, data is often messier than we expect, and scripts can contain subtle bugs. A robust system isn't just one that works when everything is perfect; it's one that behaves predictably and provides clear feedback when things go wrong.

This chapter covers how to handle errors gracefully within Computo and how to debug your scripts effectively.

2.13.1 Types of Errors

When a Computo script fails, the engine will throw a structured exception. Understanding the different types of exceptions can help you quickly diagnose the problem. The main categories are:

- **`InvalidScriptException`:** There is a fundamental structural problem with your script's JSON itself. This is rare if your JSON is well-formed.
- **`InvalidOperatorException`:** You've tried to use an operator that doesn't exist. This is almost always a typo in the operator name (e.g., `["fliter", ...]` instead of `["filter", ...]`).
- **`InvalidArgumentException`:** This is the most common category of error. It means an operator was called with the wrong number or type of arguments. Examples include:
  - `["+", 1]` (not enough arguments)
  - `["+", 1, "hello"]` (wrong argument type)
  - `["get", <object>, "/non/existent/path"]` (invalid JSON Pointer)
  - A `lambda` expression with the wrong structure.

When you run the `computo` CLI tool, it will catch these exceptions and print a descriptive error message to the console, which is your first and most important debugging tool.

2.13.2 Defensive Programming: The `if` Guard

The most common source of runtime errors is unexpected input data. A script that works perfectly with one input might fail on another if a key is missing or a value has a different type. The `if` operator is your primary tool for defensive programming.

Anti-Pattern: Assuming a Key Exists

```
["get", ["$input"], "/user/profile/name"]
```

This script will fail immediately if the `profile` key is missing from the `user` object.

Defensive Pattern: Checking for Existence A better approach is to check if the data exists before trying to access it. While Computo doesn't have a dedicated `has_key` operator, you can often achieve the same result by checking the "truthiness" of a parent object.

```
["let",
  [
    ["profile", ["get", ["$input"], "/user/profile"]]
  ],
  ["if",
    ["$", "/profile"],
    ["get", ["$", "/profile"], "/name"],
```

```
    "Default Name"
  ]
]
```

This script is more robust. It first tries to get the entire `profile` object. The `if` condition then checks if `profile` is truthy (i.e., not `null` or an empty object). Only if it's truthy does it attempt to get the `/name`. If not, it safely returns a default value.

Note: This specific pattern doesn't guard against a `profile` object that exists but is missing the `name` key. A true `has_key` or `get_with_default` operator might be added in future versions of Computo to make this even cleaner.

2.13.3 The "Debug by Deconstruction" Technique

When a complex script fails and the error message isn't immediately obvious, the best way to debug is to **deconstruct the problem**.

Imagine this complex script fails:

```
["reduce",
  ["map",
    ["filter", ["get", ["$input"], "/items"], <lambda1>],
    <lambda2>
  ],
  <lambda3>,
  0
]
```

Instead of trying to fix the whole thing at once, test each part in isolation.

1. **Test the Innermost Part:** Create a new script that *only* contains the `filter` expression. `["filter", ["get", ["$input"], "/items"], <lambda1>]` Does it work? Does it produce the array you expect? If not, the problem is in your `filter` logic.
2. **Test the Next Layer:** Once the `filter` works, wrap it in the `map`. `["map", [<working_filter_expression>], <lambda2>]` Does this produce the correct transformed array? If not, the bug is in your `map`'s lambda.
3. **Test the Final Layer:** Once the `map` and `filter` produce the correct array, use that result as the input for your `reduce` operation.

By breaking the problem down and verifying the output of each stage of your pipeline, you can methodically isolate the source of the error. Using `let` can also help here, as you can bind the result of each stage to a variable and then output that variable directly to see the intermediate data.

2.13.4 Using `permuto.apply` for Debug Output

Sometimes, the easiest way to see what's happening inside a script is to print out the state. The `permuto.apply` operator with string interpolation is a surprisingly effective debugging tool.

Let's say you're inside a `map` and you're not sure what the `user` variable contains at each step. You can insert a debugging string.

```
["map",
  ["get", ["$input"], "/users"],
  ["lambda", ["user"],
    ["permuto.apply",
      {
        "debug_message": "Processing user: ${/name} with plan: ${/plan}",
        "original_user_obj": "${/}"
      },
      ["$", "/user"]
    ]
  ]
]
```

Running this with the `--interpolation` flag will produce an array of objects, where each object is a snapshot of the `user` variable at that point in the iteration. This can be invaluable for understanding why a filter isn't working or why a value isn't what you expect. The `${/}` pointer is particularly useful, as it resolves to the entire context object.

2.13.5 In This Chapter

You've learned practical strategies for making your scripts robust and easier to debug. * You can diagnose issues by understanding Computo's **structured exception types**. * You can write **defensive scripts** that anticipate missing data by using `if` guards. * You know how to use the **"Debug by Deconstruction"** technique to methodically isolate bugs in complex pipelines. * You can use **`permuto.apply` as a "print statement"** to inspect the state of your variables during execution.

Writing code that handles failure gracefully is just as important as writing code that handles success. These techniques will help you build reliable, production-grade transformations.

2.14 Chapter 14: Best Practices and Patterns

Congratulations! You have journeyed through the entire Computo and Permuto ecosystem, from basic operators to complex data pipelines. You have all the tools you need to tackle sophisticated JSON transformation challenges.

This final chapter serves as a summary of the key strategies and best practices we've discussed. Think of it as a concise checklist to consult when you're designing and writing your own scripts. Following these principles will help you create transformations that are not just correct, but also readable, maintainable, and efficient.

2.14.1 1. Separate Logic from Presentation

This is the philosophical core of the Computo/Permuto toolkit. * **Use Computo for the "How":** Logic, conditionals, iteration, calculation, and data preparation. * **Use Permuto for the "What":** Declarative, whole-structure templating and string construction. * **The Bridge:** Use Computo to build the perfect, clean context object and then pass that clean object to `permuto.apply`. This decouples your final output structure from your messy input structure.

2.14.2 2. `let` is Your Most Valuable Tool

Proper use of `let` is the key to writing clean and efficient scripts. * **DRY (Don't Repeat Yourself):** If you use any expression more than once, bind it to a variable with `let`. * **Readability:** Use `let` to give descriptive names to complex, nested expressions. A variable named `active_premium_users` is much clearer than a multi-line `filter` expression. * **Performance:** Binding a value with `let` ensures it is evaluated only once. This is the single most important optimization technique in Computo.

2.14.3 3. Filter Early, Map Late

When building an array processing pipeline, the order of operations is critical for performance. * **Always `filter` first.** Reduce the size of your dataset at the earliest possible moment. There is no reason to `map` over and transform items that you are just going to discard. * Perform your `map` operations on the smallest possible array. This minimizes processing time and memory used for temporary objects.

2.14.4 4. Be Defensive About Input

Never assume your input data will be perfect. * **Guard your `get` calls.** Before accessing a nested value like `"/user/profile/name"`, first check if the parent (`"/user/profile"`) exists and is truthy using an `if` statement. * Provide sensible defaults in your `else` branches to handle cases of missing or `null` data. This makes your transformations more resilient and prevents unexpected failures.

2.14.5 5. Compose, Don't Build Monoliths

Computo's power comes from composing simple, single-purpose operators. * **Embrace the Pipeline:** Think of your transformation as a series of small steps (e.g., filter -> map -> reduce). This is easier to read, debug, and reason about than a single, massive, deeply nested expression. * **Recognize Patterns:** Look for opportunities to use the patterns we've discussed: * **Enrichment:** Adding new fields to an object with `merge`. * **Forking:** Creating multiple outputs from a single source. * **Aggregation:** Summarizing a list into a single report object.

2.14.6 6. Know When to Use Each Tool

Choosing the right operator or pattern for the job is a sign of mastery.

If you need to...	Your first thought should be...
Create a new object from pieces	`obj`
"Fill in the blanks" of a structure	`permuto.apply`
Make a decision	`if`
Transform every item in a list	`map`
Select a subset of items from a list	`filter`
Combine a list into a single value	`reduce`
Reuse a value or improve readability	`let`
Combine two or more objects	`merge`

2.14.7 A Final Thought

The goal of Computo and Permuto is to move complex transformation logic out of your imperative application code and into a declarative, testable, and safe data format. By treating your transformations as data, you gain clarity and flexibility.

You are now fully equipped to apply these principles to your own projects. Happy transforming

2.15 Chapter 15: Multiple Input Processing and JSON Patch Operations

In this chapter, we'll explore Computo's advanced capabilities for working with multiple input documents and performing standardized JSON diff/patch operations. These features enable sophisticated data comparison, versioning, and multi-document transformations while maintaining RFC 6902 compliance.

2.15.1 The `$inputs` System Variable: Working with Multiple Documents

Up until now, we've been working with single input documents using the `$input` variable. Computo now supports processing multiple input files simultaneously through the `$inputs` system variable.

2.15.1.1 Basic Multiple Input Usage The `$inputs` variable returns an array containing all input documents provided on the command line:

```
# Single input (traditional)
computo script.json input1.json

# Multiple inputs (new capability)
computo script.json input1.json input2.json input3.json
```

```
// Access all inputs as an array
["$inputs"]

// Access specific inputs by index
["get", ["$inputs"], "/0"]  // First input
["get", ["$inputs"], "/1"]  // Second input
["get", ["$inputs"], "/2"]  // Third input
```

2.15.1.2 Backward Compatibility The familiar `$input` variable remains fully supported and is equivalent to accessing the first input:

```
// These are equivalent:
["$input"]
["get", ["$inputs"], "/0"]
```

2.15.2 Practical Multiple Input Examples

2.15.2.1 Example 1: Merging User Profiles Let's say you have user data from two different systems that need to be merged:

profile1.json:

```
{
  "id": "user123",
  "name": "Alice Johnson",
  "last_seen": "2024-01-15T10:30:00Z"
}
```

profile2.json:

```
{
  "id": "user123",
  "email": "alice@example.com",
  "preferences": {
    "theme": "dark",
    "notifications": true
  }
}
```

merge_profiles.json:

```
["let", [
    ["profile1", ["get", ["$inputs"], "/0"]],
    ["profile2", ["get", ["$inputs"], "/1"]]
  ],
  ["obj",
    ["user_id", ["get", ["$", "/profile1"], "/id"]],
    ["name", ["get", ["$", "/profile1"], "/name"]],
    ["email", ["get", ["$", "/profile2"], "/email"]],
    ["preferences", ["get", ["$", "/profile2"], "/preferences"]],
    ["last_seen", ["get", ["$", "/profile1"], "/last_seen"]]
  ]
]
```

Usage:

```
computo --pretty=2 merge_profiles.json profile1.json profile2.json
```

Output:

```
{
  "user_id": "user123",
  "name": "Alice Johnson",
  "email": "alice@example.com",
  "preferences": {
    "theme": "dark",
    "notifications": true
  },
  "last_seen": "2024-01-15T10:30:00Z"
}
```

2.15.2.2 Example 2: Cross-Document Validation

Check if documents are consistent across different sources:

validate_consistency.json:

```
["let", [
    ["doc1", ["get", ["$inputs"], "/0"]],
    ["doc2", ["get", ["$inputs"], "/1"]],
    ["user_id_match", ["==",
      ["get", ["$", "/doc1"], "/id"],
```

```
      ["get", ["$", "/doc2"], "/id"]
    ]]
  ],
  ["obj",
    ["documents_consistent", ["$", "/user_id_match"]],
    ["doc1_id", ["get", ["$", "/doc1"], "/id"]],
    ["doc2_id", ["get", ["$", "/doc2"], "/id"]],
    ["total_inputs", ["count", ["$inputs"]]]
  ]
]
```

2.15.3 JSON Patch Operations: Diff and Patch

Computo implements RFC 6902 JSON Patch standard, enabling precise document comparison and modification through standardized operations.

2.15.3.1 The `diff` Operator Generates a JSON Patch array that describes the differences between two documents:

```
["diff", <original_document>, <modified_document>]
```

Example:

```
["diff",
  {"status": "active", "id": 123},
  {"status": "archived", "id": 123}
]
```

Output:

```
[{"op": "replace", "path": "/status", "value": "archived"}]
```

2.15.3.2 The `patch` Operator Applies a JSON Patch array to a document:

```
["patch", <document_to_modify>, <patch_array>]
```

Example:

```
["patch",
  {"status": "active", "id": 123},
  [{"op": "replace", "path": "/status", "value": "archived"}]
]
```

Output:

```
{"status": "archived", "id": 123}
```

2.15.4 Complete Diff/Patch Workflow Example

Let's walk through a complete workflow that demonstrates document versioning and change management.

2.15.4.1 Step 1: Create a Transformation Script archive_user.json:

```
["obj",
  ["id", ["get", ["$input"], "/id"]],
  ["name", ["get", ["$input"], "/name"]],
  ["status", "archived"],
  ["archived_at", "2024-01-15T12:00:00Z"]
]
```

2.15.4.2 Step 2: Generate a Patch Using CLI original_user.json:

```
{
  "id": 123,
  "name": "Alice",
  "status": "active"
}
```

Generate the patch:

```
computo --diff archive_user.json original_user.json > archive_patch.json
```

archive_patch.json (generated):

```
[
  {"op": "replace", "path": "/status", "value": "archived"},
  {"op": "add", "path": "/archived_at", "value": "2024-01-15T12:00:00Z"}
]
```

2.15.4.3 Step 3: Create a Reusable Patch Application Script apply_patch.json:

```
["patch",
  ["get", ["$inputs"], "/0"],
  ["get", ["$inputs"], "/1"]
]
```

2.15.4.4 Step 4: Apply the Patch

```
computo --pretty=2 apply_patch.json original_user.json archive_patch.json
```

Output:

```
{
  "id": 123,
  "name": "Alice",
  "status": "archived",
  "archived_at": "2024-01-15T12:00:00Z"
}
```

2.15.5 Advanced Multi-Document Processing Patterns

2.15.5.1 Pattern 1: Document Synchronization Compare configurations between environments and generate sync patches:

sync_configs.json:

```
["let", [
    ["prod_config", ["get", ["$inputs"], "/0"]],
    ["staging_config", ["get", ["$inputs"], "/1"]],
    ["sync_patch", ["diff", ["$", "/prod_config"], ["$", "/staging_config"]]]
  ],
  ["obj",
    ["requires_sync", [">", ["count", ["$", "/sync_patch"]], 0]],
    ["patch_operations", ["$", "/sync_patch"]],
    ["staging_after_sync", ["if",
      ["$", "/requires_sync"],
      ["patch", ["$", "/staging_config"], ["$", "/sync_patch"]],
      ["$", "/staging_config"]
    ]]
  ]
]
```

2.15.5.2 Pattern 2: Version Rollback Generation

Create rollback patches by reversing the diff direction:

generate_rollback.json:

```
// Generate rollback patch (reverse diff)
["diff",
  ["get", ["$inputs"], "/1"],  // new version
  ["get", ["$inputs"], "/0"]   // original version
]
// This creates a patch that rolls back from new to original
```

2.15.5.3 Pattern 3: Multi-Source Data Aggregation

Combine data from multiple APIs or databases:

aggregate_user_data.json:

```
["let", [
    ["profile_service", ["get", ["$inputs"], "/0"]],
    ["settings_service", ["get", ["$inputs"], "/1"]],
    ["activity_service", ["get", ["$inputs"], "/2"]]
  ],
  ["obj",
    ["user_id", ["get", ["$", "/profile_service"], "/id"]],
    ["basic_info", ["obj",
      ["name", ["get", ["$", "/profile_service"], "/name"]],
      ["email", ["get", ["$", "/profile_service"], "/email"]]
    ]],
    ["preferences", ["get", ["$", "/settings_service"], "/preferences"]],
    ["recent_activity", ["get", ["$", "/activity_service"], "/last_actions"]],
    ["last_updated", ["get", ["$", "/activity_service"], "/timestamp"]]
```

```
  ]
]
```

2.15.6 Error Handling for Patch Operations

When working with patch operations, be aware of potential failures:

2.15.6.1 Common Patch Failures

1. **Invalid patch format** - Patch must be a valid JSON array
2. **Failed test operations** - `test` operations that don't match expected values
3. **Invalid paths** - Attempting to modify non-existent paths
4. **Malformed operations** - Invalid operation types

2.15.6.2 Safe Patch Application Pattern Use conditional logic to handle potential patch failures gracefully:

safe_patch_apply.json:

```
["let", [
    ["document", ["get", ["$inputs"], "/0"]],
    ["patch_ops", ["get", ["$inputs"], "/1"]],
    ["patch_count", ["count", ["$", "/patch_ops"]]]
  ],
  ["obj",
    ["original_document", ["$", "/document"]],
    ["patch_operations", ["$", "/patch_ops"]],
    ["patch_safe", ["==", ["$", "/patch_count"], 1]],
    ["result", ["if",
      ["$", "/patch_safe"],
      ["patch", ["$", "/document"], ["$", "/patch_ops"]],
      ["$", "/document"]
    ]]
  ]
]
```

2.15.7 CLI Features for Diff/Patch Operations

2.15.7.1 The `--diff` Flag Generate patches directly from transformations without modifying your scripts:

```
# Traditional transformation
computo transform.json input.json

# Generate patch from same transformation
computo --diff transform.json input.json
```

This is particularly useful for: - **Version control integration** - Generate patches for change tracking - **Automated deployment** - Create configuration update patches - **Data migration planning** - Preview changes before applying them

2.15.7.2 Combining with Multiple Inputs

Remember that `--diff` only works with a single input file:

```
# Valid: Single input with --diff
computo --diff script.json input.json

# Invalid: Multiple inputs with --diff
computo --diff script.json input1.json input2.json
```

2.16 Functional List Processing with car and cdr

Computo includes functional programming operators `car` and `cdr` inspired by Lisp, which provide elegant ways to work with arrays and multiple inputs.

2.16.1 Understanding car and cdr

```
// car: Get the first element
["car", {"array": [1, 2, 3, 4]}]
// Result: 1

// cdr: Get everything except the first element
["cdr", {"array": [1, 2, 3, 4]}]
// Result: [2, 3, 4]

// Composition: Get the second element
["car", ["cdr", {"array": [1, 2, 3, 4]}]]
// Result: 2
```

2.16.2 Functional Multiple Input Processing

The `car` and `cdr` operators are particularly powerful for processing multiple inputs in a functional style:

2.16.2.1 Example: Applying Multiple Patches

Traditional approach:

```
["let", [
    ["initial", ["get", ["$inputs"], "/0"]],
    ["patch1", ["get", ["$inputs"], "/1"]],
    ["patch2", ["get", ["$inputs"], "/2"]]
  ],
  ["patch", ["patch", ["$", "/initial"], ["$", "/patch1"]], ["$", "/patch2"]]
]
```

Functional approach with car/cdr:

```
["reduce",
  ["cdr", ["$inputs"]],                    // All patches (skip first input)
  ["lambda", ["state", "patch"],
    ["patch", ["$", "/state"], ["$", "/patch"]]
  ],
```

```
  ["car", ["$inputs"]]                    // Initial state (first input)
]
```

Benefits of the functional approach: - Works with any number of patches (not just 2) - More readable and declarative - Follows functional programming principles - Easier to test and reason about

2.16.2.2 Example: Processing Conversation Updates

```
// Process conversation updates using functional list operations
["let", [
    ["initial_conversation", ["car", ["$inputs"]]],     // First input
    ["all_patches", ["cdr", ["$inputs"]]],              // Remaining inputs
    ["final_state", ["reduce",
      ["$", "/all_patches"],
      ["lambda", ["conversation", "patch"],
        ["patch", ["$", "/conversation"], ["$", "/patch"]]
      ],
      ["$", "/initial_conversation"]
    ]],
    ["patch_count", ["count", ["$", "/all_patches"]]]
  ],
  ["obj",
    ["conversation_id", ["get", ["$", "/final_state"], "/id"]],
    ["message_count", ["count", ["get", ["$", "/final_state"], "/messages"]]],
    ["patches_applied", ["$", "/patch_count"]],
    ["final_conversation", ["$", "/final_state"]]
  ]
]
```

Usage:

```
computo --pretty=2 conversation_processor.json initial_conversation.json patch1.json patch2.jso
```

2.16.3 Best Practices for Multiple Input Processing

1. **Use descriptive variable names** when working with `let` bindings
2. **Validate input count** using `["count", ["$inputs"]]` when expected
3. **Handle missing inputs gracefully** with conditional logic
4. **Document input order requirements** in your scripts
5. **Use patch operations for incremental updates** rather than full replacements
6. **Consider car/cdr for functional processing** when dealing with variable numbers of inputs
7. **Use car/cdr with reduce** for applying operations across multiple inputs

2.16.4 Chapter Summary

In this chapter, you learned:

- How to process multiple input documents using the `$inputs` system variable
- The difference between `$input` (first document) and `$inputs` (all documents)
- How to generate standardized JSON patches using the `diff` operator

- How to apply patches using the `patch` operator

- **Functional list processing** with `car` and `cdr` operators for elegant array manipulation
- **Functional patterns** for processing variable numbers of inputs
- Complete workflows for document versioning and change management
- Advanced patterns for multi-document processing
- Error handling strategies for patch operations
- CLI features for streamlined diff/patch workflows

These features enable Computo to handle complex scenarios involving document comparison, versioning, configuration management, and multi-source data processing while maintaining RFC 6902 compliance for interoperability with other JSON Patch tools.

In the next chapter, we'll explore performance optimization techniques and best practices for production deployments.

2.17 Appendix A: Complete Operator Reference

This appendix provides a quick reference for all 27 operators available in Computo.

2.17.1 Data Access & Scoping

2.17.1.1 `let` Binds variables to expressions for use within a scoped body. * **Syntax:** `["let", [["var1", <expr1>], ...], <body_expr>]` * **Example:** `["let", [["x", 5]], ["+", ["$", "/x"], 10]]` -> `15`

2.17.1.2 `$` Retrieves the value of a variable defined by `let`. * **Syntax:** `["$", "/variable_name"]` * **Example:** `["let", [["x", 5]], ["$", "/x"]]` -> `5`

2.17.1.3 `get` Retrieves a value from a JSON object or array using a JSON Pointer. * **Syntax:** `["get", <object_expr>, <json_pointer_string>]` * **Example:** `["get", {"obj": {"a": 1}}, "/obj/a"]` -> `1`

2.17.1.4 `$input` Returns the entire input JSON document that was provided to the script (equivalent to the first input when multiple inputs are provided). * **Syntax:** `["$input"]` * **Example:** If `input.json` is `{"id": 1}`, `["$input"]` -> `{"id": 1}`

2.17.1.5 `$inputs` Returns an array containing all input JSON documents that were provided to the script. * **Syntax:** `["$inputs"]` * **Example:** If called with `input1.json` and `input2.json`, returns `[<input1_content>, <input2_content>]`

2.17.2 Logic & Control Flow

2.17.2.1 `if` Evaluates a condition and returns one of two expressions. * **Syntax:** `["if", <condition>, <then_expr>, <else_expr>]` * **Example:** `["if", true, "Yes", "No"]` -> `"Yes"`

2.17.2.2 `==`, `!=`, `>`, `<`, `>=`, `<=` Perform a comparison between two values. Math comparisons require numbers. * **Syntax:** `["<op>", <expr1>, <expr2>]` * **Example:** `[">", 10, 5]` -> `true` * **Example:** `["==", "a", "a"]` -> `true`

2.17.2.3 `approx` Checks if two numbers are approximately equal within a given epsilon. * **Syntax:** `["approx", <num1_expr>, <num2_expr>, <epsilon_expr>]` * **Example:** `["approx", 3.14, 3.141, 0.01]` -> `true`

2.17.3 Data Construction & Manipulation

2.17.3.1 `obj` Creates a JSON object from key-value pairs. * **Syntax:** `["obj", ["key1", <val1>], ["key2", <val2>], ...]` * **Example:** `["obj", ["a", 1], ["b", 2]]` -> `{"a": 1, "b": 2}`

2.17.3.2 `merge` Merges two or more objects. Rightmost keys win in case of conflict. * **Syntax:** `["merge", <obj1_expr>, <obj2_expr>, ...]` * **Example:** `["merge", {"a": 1, "b": 1}, {"b": 2, "c": 3}]` -> `{"a": 1, "b": 2, "c": 3}`

2.17.3.3 `permuto.apply` Applies a Permuto template to a context object. * **Syntax:** `["permuto.apply", <template_expr>, <context_expr>]` * **Example:** `["permuto.apply", {"id": "${/user_id}"}, {"user_id": 123}]` -> `{"id": 123}`

2.17.4 JSON Patch Operations (RFC 6902)

2.17.4.1 `diff` Generates an RFC 6902 JSON Patch array that describes the differences between two JSON documents. * **Syntax:** `["diff", <original_document>, <modified_document>]` * **Example:** `["diff", {"status": "active"}, {"status": "archived"}]` -> `[{"op": "replace", "path": "/status", "value": "archived"}]`

2.17.4.2 `patch` Applies an RFC 6902 JSON Patch array to a document, returning the modified document. * **Syntax:** `["patch", <document_to_patch>, <patch_array>]` * **Example:** `["patch", {"status": "active"}, [{"op": "replace", "path": "/status", "value": "archived"}]]` -> `{"status": "archived"}`

2.17.5 Mathematical

2.17.5.1 `+`, `-`, `*`, `/` Performs a mathematical operation on two numbers. * **Syntax:** `["<op>", <num1_expr>, <num2_expr>]` * **Example:** `["*", 5, 10]` -> `50`

2.17.6 Array Operators

Note: For all array operators, literal arrays must be specified using `{"array": [...]}` syntax.

2.17.6.1 `map` Applies a lambda to each item in an array, returning a new array of transformed items. * **Syntax:** `["map", <array_expr>, ["lambda", ["item_var"], <transform_expr>]]` * **Example:** `["map", {"array": [1,2]}, ["lambda",["x"],["+",["$","/x"],1]]]` -> `[2, 3]`

2.17.6.2 `filter` Returns a new array containing only items for which the lambda returns a truthy value. * **Syntax:** `["filter", <array_expr>, ["lambda", ["item_var"], <condition_expr>]]` * **Example:** `["filter", {"array": [0,1,2]}, ["lambda",["x"],["$","/x"]]]` -> `[1, 2]`

2.17.6.3 `reduce` Reduces an array to a single value by applying a two-argument lambda. * **Syntax:** `["reduce", <array_expr>, ["lambda", ["acc", "item"], <expr>], <initial_value>]` * **Example:** `["reduce", {"array": [1,2,3]}, ["lambda",["a","i"],["+",["$","/a"],0]]` -> `6`

2.17.6.4 `find` Returns the **first item** in an array for which the lambda returns a truthy value, or `null` if no match is found. * **Syntax:** `["find", <array_expr>, ["lambda", ["item_var"], <condition_expr>]]` * **Example:** `["find", {"array": [1,5,10]}, ["lambda",["x"],[">",["$","/x"],3]]]` -> `5`

2.17.6.5 `some` Returns `true` if the lambda returns a truthy value for **at least one item** in the array, `false` otherwise. * **Syntax:** `["some", <array_expr>, ["lambda", ["item_var"], <condition_expr>]]` * **Example:** `["some", {"array": [1,5,10]}, ["lambda",["x"],["==",["$","/x"],5]]]` -> `true`

2.17.6.6 `every` Returns `true` if the lambda returns a truthy value for **all items** in the array, `true` for an empty array, `false` otherwise. * **Syntax:** `["every", <array_expr>, ["lambda", ["item_var"], <condition_expr>]]` * **Example:** `["every", {"array": [2,4,6]}, ["lambda",["x"],["==",["%",["$","/x"],2],0]]]` -> `true`

2.17.6.7 `flatMap` Like `map`, but if a lambda returns an array, its items are flattened into the result array. * **Syntax:** `["flatMap", <array_expr>, ["lambda", ["item_var"], <transform_expr>]]` * **Example:** `["flatMap", {"array": [1,2]}, ["lambda",["x"],{"array":[["$","/x"` -> `[1, 1, 2, 2]`

2.17.6.8 `count` Returns the number of items in an array. * **Syntax:** `["count", <array_expr>]` * **Example:** `["count", {"array": [1, 2, 3]}]` -> `3`

2.17.7 List Processing (Functional)

2.17.7.1 `car` Returns the first element of an array. Inspired by Lisp's car function. * **Syntax:** `["car", <array_expr>]` * **Example:** `["car", {"array": [1, 2, 3]}]` -> `1` * **Note:** Throws `InvalidArgumentException` if array is empty or argument is not an array

---

2.17.7.2 `cdr` Returns all elements except the first from an array. Inspired by Lisp's cdr function. * **Syntax:** `["cdr", <array_expr>]` * **Example:** `["cdr", {"array": [1, 2, 3]}]` -> `[2, 3]` * **Note:** Returns empty array `[]` if input array is empty; throws `InvalidArgumentException` if argument is not an array