Multiplication Strategies: Commutative Reasoning

Compiled by: Theodore M. Savich

March 11, 2025

Commutative Action for Multiplication

Imagine a situation where we have six chocolate chip cookies with 4 chocolate chips in each cookie. That's 24 chocolate chips. Instead, we imagine we have four chocolate chip cookies, and each cookie has 6 chocolate chips. That's still 24 chocolate chips, but not enough cookies to feed 4 kids! The commutative property of multiplication,

- **Definition:** For any two natural numbers a and b,

$$a \times b = b \times a.$$

- **Example:** $3 \times 4 = 4 \times 3$.

is fine for purely abstract mathematical contexts, but in *equal groups multiplication problems* – the sort of problems that most people encounter when learning about multiplication for the first time – the order of the factors can make a big difference.

However, there is a big difference between recognizing the commutative property holds for the number of chocolate chips and the fact that you would have two crying kids if there were 6 kids and you only had 4 cookies.

For equal groups multiplication:

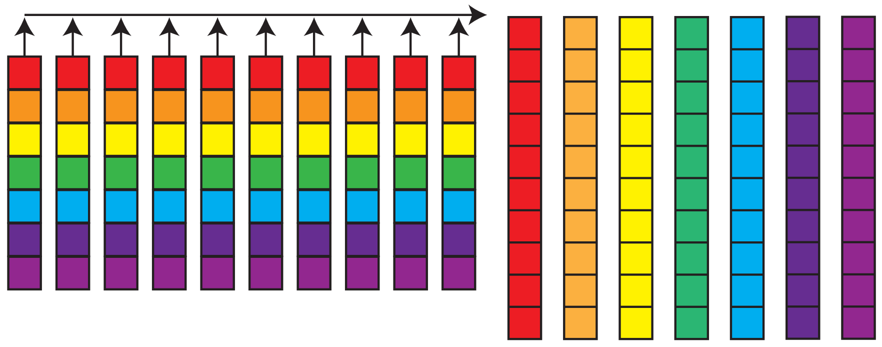
$$\boxed{\text{number of groups}} \times \boxed{\text{number of items in each group}} = \boxed{\text{total number of items}}$$

To act (or reason) commutatively, the number of items in a group needs to be repackaged as the number of groups, while the number of groups transforms into the number of items in each group.

Definition and Example

Imagine this situation, from Hackenberg (2025): we have ten packages of rainbow flavored candies, and each package contains the 7 candies, one of each of the 'colors of the rainbow': red, orange, yellow, green, blue, indigo, and violet.

It can be hard to count 7 objects ten times. $7 + 7 + 7 + 7 + 7 + 7 + 7 + 7 + 7 + 7$ is a lot of work! We can *repackage* the candies into seven packages of 10 candies each - all the reds together, all the oranges together, and so on. The result is seven 10s, which is a lot easier to count: $10 + 10 + 10 + 10 + 10 + 10 + 10 = 70$.



Objective of the Automaton

- **Input:** A multiplication expression $a \times b$.
- **Output:** The transformed expression $b \times a$.
- **Functionality:** Recognize when a multiplication expression is presented and apply the commutative property to reorder the operands.

Automaton Type Selection

Finite State Transducer (FST)

- **Transduction Capability:** Unlike finite state automata (FSA) that merely recognize languages, an FST can transform input strings into output strings.
- **Suitability:** Ideal for tasks involving input-output transformations, such as repackaging operands in a multiplication expression.

Designing the FST for Commutative Reasoning

Components of the FST

1. **States** (Q):
   - q_0: Start state.
   - q_1: Reading the first operand.
   - q_2: Reading the multiplication symbol ($\times$).
   - q_3: Reading the second operand.
   - q_4: Applying the commutative transformation.
   - q_{accept}: Accepting state; transformation complete.
2. **Input Alphabet** (Σ):
   - Digits: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
   - Multiplication symbol: $\times$
3. **Output Alphabet** (Δ):
   - Digits: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

- Multiplication symbol: $\times$

4. **Transition Function (δ):** Defines how the FST transitions between states based on input symbols and produces corresponding output symbols.
5. **Start State:** q_0
6. **Accepting State:** q_{accept}

Transition Function Details (Single-Digit Operands)

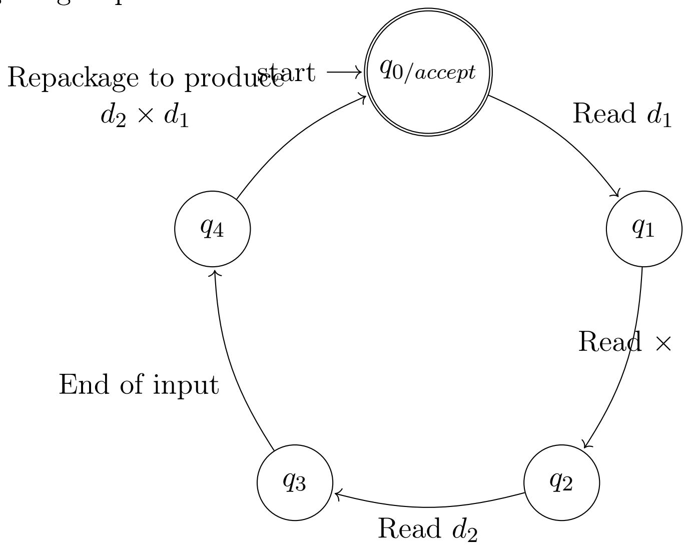
For simplicity, assume operands are single digits. The FST behaves as follows:

Current State	Input Symbol	Read Symbol	Next State	Output Symbol
q_0	Any digit d_1	d_1	q_1	d_1
q_1	$\times$	$\times$	q_2	$\times$
q_2	Any digit d_2	d_2	q_3	d_2
q_3	End of input	—	q_4	—
q_4	—	—	q_{accept}	Output repackaged expression: $d_2 \times d_1$

Automaton Diagrams

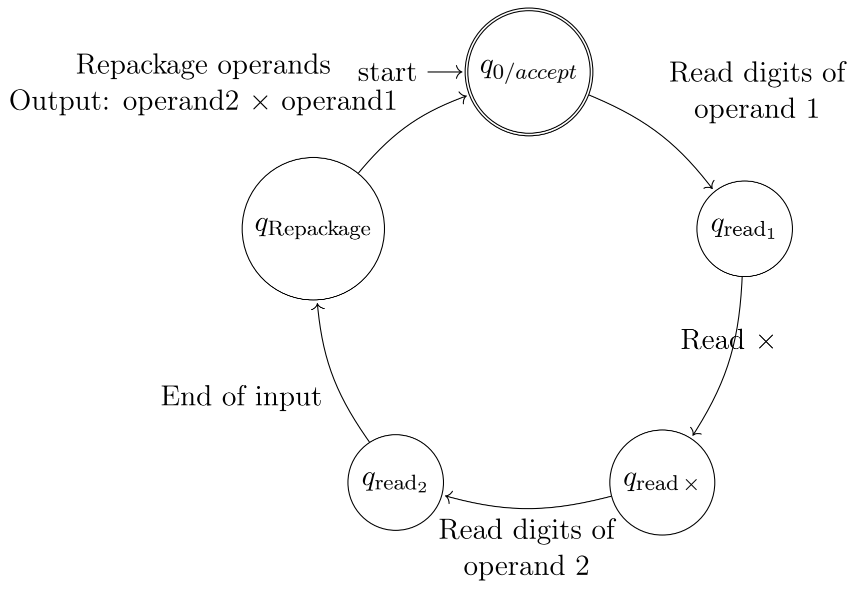
Circular Diagram for Single-Digit Operands

Below is the circular state diagram for the FST (with the start and accept states merged) for single-digit operands.



Circular Diagram for Multi-Digit Operands

For multi-digit operands, the FST buffers digits until the entire operand is read, then repackages the operands. The following circular diagram represents an enhanced FST:



Example Execution

Problem:

3×4

Execution Steps:

1. q_0: Reads the digit '3', outputs '3', then moves to q_1.
2. q_1: Reads '×', outputs '×', then moves to q_2.
3. q_2: Reads the digit '4', outputs '4', then moves to q_3.
4. q_3: End of input is detected; transition to q_4.
5. q_4: Repackages the operands to produce '4 × 3', and transitions back to $q_{0/accept}$ (accepting state).

Output:

4×3

Conclusion

By designing this Finite State Transducer (FST), we effectively model the commutative property of multiplication as a transformation process. The single-digit version demonstrates the basic concept, while the multi-digit version shows how the automaton can be extended to handle more complex expressions by buffering entire operands before applying the repackage.

HTML Implementation

```html
<!DOCTYPE html>
<html>
<head>
    <title>Commutative Multiplication</title>
<style>
    body { font-family: sans-serif; }
    .cube-row { display: flex; margin-bottom: 2px; } /* Arrange cubes in a row */
    .cube {
        width: 15px; /* Cube size */
        height: 15px;
        border: 1px solid #ccc; /* Cube border */
        margin-right: 2px; /* Spacing between cubes */
        display: inline-block; /* Ensure inline display for flexbox */
    }
    /* Rainbow colors for cubes - you can customize these */
    .cube.red { background-color: red; }
    .cube.orange { background-color: orange; }
    .cube.yellow { background-color: yellow; }
    .cube.green { background-color: green; }
    .cube.blue { background-color: blue; }
    .cube.indigo { background-color: indigo; }
    .cube.violet { background-color: violet; }

</style>
</head>
<body>
    <h1>Commutative Reasoning for Multiplication</h1>

    <div>
        <label for="commuteA">Factor 1:</label>
        <input type="number" id="commuteA" value="10">
    </div>
    <div>
        <label for="commuteB">Factor 2:</label>
        <input type="number" id="commuteB" value="7">
    </div>

    <button onclick="runCommutativeAutomaton()">Repackage and Visualize</button>

    <div id="commuteOutput">
        <!-- Output will be displayed here -->
    </div>

    <script>
        document.addEventListener('DOMContentLoaded', function() {
            const commuteOutputElement = document.getElementById('commuteOutput');
            const commuteAInput = document.getElementById('commuteA');
            const commuteBInput = document.getElementById('commuteB');

            window.runCommutativeAutomaton = function() {
                try {
                    const factorA = commuteAInput.value;
```

```javascript
            const factorB = commuteBInput.value;

            if (isNaN(parseInt(factorA)) || isNaN(parseInt(factorB)) || parseInt(factorA) <= 0 || parseInt(factorB) <= 0) {
                commuteOutputElement.textContent = "Please enter valid positive numbers for both factors";
                return;
            }

            let output = '';
            output += `<h2>Commutative Repackaging for Multiplication</h2>\n\n`;
            output += `<p><strong>Original Expression:</strong> ${factorA} &times; ${factorB}</p>\n`; // Updated to display the multiplication symbol correctly

            // --- Simulate FST Transformation ---
            const transformedFactorA = factorB;
            const transformedFactorB = factorA;

            output += `<p><strong>Applying Commutative Repackaging...</strong></p>\n`;
            output += `<p>We transform the expression by swapping the order of the factors.</p>\n`;
            output += `<p><strong>Repackaged Expression:</strong> ${transformedFactorA} &times; ${transformedFactorB}</p>\n\n`;

            // --- Visualize with Colorful Cubes ---
            const numFactorA = parseInt(factorA);
            const numFactorB = parseInt(factorB);
            const productAB = numFactorA * numFactorB;
            const productBA = parseInt(transformedFactorA) * parseInt(transformedFactorB);

            output += `<p><strong>Visualizing the Repackaging:</strong></p>\n`;

            // Arrangement 1 (Original: A x B) - Cubes
            output += `<p><strong>Arrangement 1: ${factorA} groups of ${factorB} items each</strong></p>\n`;
            output += `<p>Visual representation:</p>\n`;
            for (let i = 0; i < numFactorA; i++) {
                output += `<div class='cube-row'>`; // Start a new row for cubes
                for (let j = 0; j < numFactorB; j++) {
                    const rainbowColors = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet'];
                    const colorClass = rainbowColors[j % rainbowColors.length]; // Cycle through rainbow colors
                    output += `<span class='cube ${colorClass}'></span>`; // Create a cube with color class
                }
                output += `</div>`; // End the cube row
            }
            output += `<p>Total: ${productAB} items</p>\n\n`;

            // Arrangement 2 (Repackaged: B x A) - Cubes
```

```
                    output += `<p><strong>Arrangement 2: ${transformedFactorA} groups of ${transformedFactorB} items each</strong></p>\n`;
                    output += `<p>Visual representation:</p>\n`;
                    for (let i = 0; i < parseInt(transformedFactorA); i++) {
                        output += `<div class='cube-row'>`; // Start a new row
                        for (let j = 0; j < parseInt(transformedFactorB); j++) {
                            const rainbowColors = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet'];
                            const colorClass = rainbowColors[j % rainbowColors.length];
                            output += `<span class='cube ${colorClass}'></span>`; // Create colored cube
                        }
                        output += `</div>`; // End row
                    }
                    output += `<p>Total: ${productBA} items</p>\n\n`;


                    output += `<p><strong>Conclusion:</strong></p>\n`;
                    output += `<p>By commutatively repackaging ${factorA} &times; ${factorB} into ${transformedFactorA} &times; ${transformedFactorB}, we change the grouping but maintain the same total quantity (${productAB} = ${productBA}).</p>\n`;


                    commuteOutputElement.innerHTML = output;


                } catch (error) {
                    commuteOutputElement.textContent = `Error: ${error.message}`;
                }
            };
        });
    </script>
</body>
</html>
```