Understanding the Node Classes Line by Line

Let me walk you through the base `Node` class and `ScriptNode` class in detail:

Base `Node` Class

```python
class Node:
    """Base class for all syntax tree nodes"""
    def __init__(self):
        self.result = None

    def evaluate(self, executor_func, verify_hash=False, script_hashes=None):
        """To be implemented by subclasses"""
        raise NotImplementedError
```

Let's break it down:

1. `class Node:` - Defines a base class called `Node` that all other node types will inherit from.

2. `"""Base class for all syntax tree nodes"""` - A docstring explaining the purpose of this class.

3. `def __init__(self):` - The constructor method that's called when creating a new `Node` object.

4. `self.result = None` - Initializes a variable `result` to store the evaluation result, starting with `None`.

5. `def evaluate(self, executor_func, verify_hash=False, script_hashes=None):` - Defines a method that each subclass must implement.
   - `executor_func` - A function that will run scripts
   - `verify_hash` - A flag to indicate whether to verify script directory hashes
   - `script_hashes` - A dictionary mapping script names to their expected hashes

6. `raise NotImplementedError` - Forces subclasses to provide their own implementation of this method.

ScriptNode Class

```python
class ScriptNode(Node):
    """Node representing a script execution"""
    def __init__(self, name, args=None):
        super().__init__()
```

```
        self.name = name
        self.args = args or []

    def evaluate(self, executor_func, verify_hash=False, script_hashes=None):
        """Execute the script and return result"""
        expected_hash = script_hashes.get(self.name) if verify_hash and script_hashes else None
        self.result = executor_func(self.name, self.args, verify_hash, expected_hash)

        # In logical context, 0 (success) = True, 1 (failure) = False
        logical_result = (self.result == 0)

        # Write the result code to stderr for individual script
        sys.stderr.write(f"{self.result}\n")

        return logical_result

    def __str__(self):
        if self.args:
            return f"({self.name}:{','.join(self.args)})"
        return f"({self.name})"
```

Let's examine this line by line:

1. `class ScriptNode(Node):` - Defines a class `ScriptNode` that inherits from `Node`.

2. `"""Node representing a script execution"""` - A docstring explaining this class's purpose.

3. `def __init__(self, name, args=None):` - Constructor that takes a script name and optional arguments.

4. `super().__init__()` - Calls the parent class's constructor to initialize the base class.

5. `self.name = name` - Stores the script name (like "A" or "B").

6. `self.args = args or []` - Stores script arguments, using an empty list if no arguments provided.

7. `def evaluate(self, executor_func, verify_hash=False, script_hashes=None):` - Implements the required evaluation method.

8. `expected_hash = script_hashes.get(self.name) if verify_hash and script_hashes else None` - Gets the expected hash for this script, or `None` if we're not verifying hashes.

9. `self.result = executor_func(self.name, self.args, verify_hash, expected_hash)` - Calls the provided executor function to run the script, passing the script name, arguments, verification flag, and expected hash.

10. `logical_result = (self.result == 0)` - Converts the numeric result (0 for success, 1 for failure) into a Boolean value (True for success, False for failure).

11. `sys.stderr.write(f"{self.result}\n")` - Writes the result code (0 or 1) to standard error output.

12. `return logical_result` - Returns the Boolean result (True for success, False for failure).

13. `def __str__(self):` - Defines how this object should be converted to a string (for printing).

14. `if self.args: return f"({self.name}:{','.join(self.args)})" else: return f"({self.name})"` - Returns a string representation of the script node, including arguments if any.

Example Walkthrough

Let's follow the execution of a `ScriptNode` for script "A" with arguments "hello" and "world":

1. **Creation**: `script_node = ScriptNode("A", ["hello", "world"])`
   - `self.name` becomes "A"
   - `self.args` becomes ["hello", "world"]

2. **Evaluation**: `result = script_node.evaluate(dynamic_import_and_run, True, {"A": "6c8c069a22d96be8a18c21722cdac82d"})`

   - Gets `expected_hash = "6c8c069a22d96be8a18c21722cdac82d"`
   - Calls `dynamic_import_and_run("A", ["hello", "world"], True, "6c8c069a22d96be8a18c21722cdac82d")`
   - The executor function:
     - Checks if directory A exists
     - Verifies the directory hash matches the expected hash
     - Imports A.py module
     - Runs `A("hello", "world")`
     - Returns 0 if successful, 1 if any errors
   - Let's say it returns 0 (success)
   - `self.result` becomes 0
   - `logical_result` becomes True (because 0 == 0)
   - Writes "0" to stderr
   - Returns True

3. **String Representation**: When printing this node, it would appear as `(A:hello,world)`

This design allows you to build a tree of operations (with AND, OR, and NOT nodes) that can evaluate complex logical expressions involving script executions.

Explaining to Your Manager

"We've designed a system that represents logical expressions as a tree structure. At the leaf nodes of the tree, we have 'ScriptNode' objects that represent individual scripts to run. When we evaluate the tree, each script node runs its script using the name and arguments, checks the hash if requested, and returns success or failure. This result gets passed up to the logical operator nodes, which combine the results according to their rules - AND requires all to succeed, OR requires at least one, and NOT inverts the result. This tree-based approach gives us a flexible way to express complex execution flows."

//

Understanding LogicalOperatorNode and NotNode Line by Line

Let me explain each class in detail and walk you through examples.

LogicalOperatorNode Class

```python
class LogicalOperatorNode(Node):
    """Node representing a logical operator with children"""
    def __init__(self, operator, children=None):
        super().__init__()
        self.operator = operator
        self.children = children or []

    def add_child(self, child):
        self.children.append(child)

    def __str__(self):
        children_str = ", ".join(str(child) for child in self.children)
        return f"{self.operator} [ {children_str} ]"
```

Line-by-Line Explanation:

1. `class LogicalOperatorNode(Node):` - Defines a class that inherits from the `Node` base class.

2. `"""Node representing a logical operator with children"""` - A docstring explaining the class purpose.

3. `def __init__(self, operator, children=None):` - Constructor that takes an operator symbol (like "&&" or "||") and an optional list of child nodes.

4. `super().__init__()` - Calls the parent class's constructor to initialize the base `Node` properties.

5. `self.operator = operator` - Stores the operator symbol (e.g., "&&", "||", "&", "|").

6. `self.children = children or []` - If `children` is provided, use it; otherwise, initialize with an empty list.

7. `def add_child(self, child):` - A method to add a child node to this operator's list of children.

8. `self.children.append(child)` - Adds the provided child node to the children list.

9. `def __str__(self):` - Defines how to represent this object as a string (for debugging/printing).

10. `children_str = ", ".join(str(child) for child in self.children)` - Creates a comma-separated string of all child nodes.

11. `return f"{self.operator} [ {children_str} ]"` - Returns a formatted string like "&& [(A), (B)]".

NotNode Class

```python
class NotNode(Node):
    """Node representing a NOT operator"""
    def __init__(self, child=None):
        super().__init__()
        self.child = child

    def evaluate(self, executor_func, verify_hash=False, script_hashes=None):
        """Evaluate NOT node by inverting the result of its child"""
        if self.child is None:
            print("Warning: NOT operator with no child")
            return False

        child_result = self.child.evaluate(executor_func, verify_hash, script_hashes)
        result = not child_result

        print(f"NOT operator: inverting {child_result} to {result}")
        self.result = result
        return result
```

```
    def __str__(self):
        return f"!{self.child}"
```

Line-by-Line Explanation:

1. `class NotNode(Node):` - Defines a class that inherits from `Node`.

2. `"""Node representing a NOT operator"""` - A docstring explaining the class purpose.

3. `def __init__(self, child=None):` - Constructor that takes an optional child node to negate.

4. `super().__init__()` - Calls the parent class's constructor.

5. `self.child = child` - Stores the child node that will be negated.

6. `def evaluate(self, executor_func, verify_hash=False, script_hashes=None):` - Implements the evaluation method.

7. `if self.child is None:` - Checks if there's no child node to negate.

8. `print("Warning: NOT operator with no child")` - Issues a warning if there's no child.

9. `return False` - Returns False if there's no child (safety default).

10. `child_result = self.child.evaluate(executor_func, verify_hash, script_hashes)` - Evaluates the child node.

11. `result = not child_result` - Negates the child's result (True becomes False, False becomes True).

12. `print(f"NOT operator: inverting {child_result} to {result}")` - Prints information about the negation.

13. `self.result = result` - Stores the result for potential later reference.

14. `return result` - Returns the negated result.

15. `def __str__(self):` - Defines the string representation.

16. `return f"!{self.child}"` - Returns a string like "!(A)" for a NOT node with a script child.

Key Differences Between These Classes

1. **Children Storage**:
   - `LogicalOperatorNode` can have multiple children in a list
   - `NotNode` has a single child (logical NOT has only one operand)

2. **Evaluation**:
   - `LogicalOperatorNode` doesn't implement evaluation (must be done in subclasses like `AndNode`, `OrNode`)
   - `NotNode` implements evaluation by negating its child's result

Walk-through Examples

Example 1: LogicalOperatorNode as an AndNode

```python
# Creating nodes
script_a = ScriptNode("A")
script_b = ScriptNode("B")

# Creating AND node as a logical operator
and_node = AndNode(circuit_breaking=True) # AndNode inherits from LogicalOperatorNode
and_node.add_child(script_a)
and_node.add_child(script_b)

# What happens when we print it:
print(and_node)  # Output: && [ (A), (B) ]
```

Here, `LogicalOperatorNode` is used as a base class for the `AndNode`. The `add_child` method allows us to build the tree by adding script nodes as children.

Example 2: NotNode with a Script

```python
# Create a ScriptNode
script_c = ScriptNode("C", ["test"])

# Create a NOT node that negates script_c
not_node = NotNode(script_c)

# What happens when we print it:
print(not_node)  # Output: !(C:test)

# What happens during evaluation:
```

```
# 1. If script C succeeds (returns True):
#    - child_result = True
#    - result = not True = False
#    - Prints: "NOT operator: inverting True to False"
#    - Returns: False

# 2. If script C fails (returns False):
#    - child_result = False
#    - result = not False = True
#    - Prints: "NOT operator: inverting False to True"
#    - Returns: True
```

Example 3: More Complex Tree with Both Node Types

```python
# Create script nodes
script_a = ScriptNode("A")
script_b = ScriptNode("B")
script_c = ScriptNode("C")

# Create NOT node for script_c
not_c = NotNode(script_c)

# Create an AND between A and B
and_node = AndNode(circuit_breaking=True)
and_node.add_child(script_a)
and_node.add_child(script_b)

# Create an OR between (A AND B) and (NOT C)
or_node = OrNode(circuit_breaking=True)
or_node.add_child(and_node)
or_node.add_child(not_c)

# What happens when we print it:
print(or_node)  # Output: || [ && [ (A), (B) ], !(C) ]
```

This creates a logical tree for the expression: `(A && B) || (!C)`

When evaluating this tree:
1. First, `A && B` is evaluated
   - If both succeed, the result is `True`
   - If either fails, the result is `False`

2. In parallel, `!C` is evaluated
   - If C succeeds (returns `True`), `!C` returns `False`
   - If C fails (returns `False`), `!C` returns `True`
3. Finally, the OR combines these results
   - If either side is `True`, the final result is `True`
   - Only if both sides are `False` is the final result `False`

Summary for Your Manager

"The `LogicalOperatorNode` class is a base class that represents logical operations like AND and OR. It can have multiple children and provides methods to build the expression tree. The `NotNode` class represents the NOT operator that inverts the result of its child node. It evaluates by running its child and then negating the result.

For example, if we build an expression like '!(A) && B', the NOT node inverts A's result before combining it with B using AND. This gives us the full power of Boolean logic when defining which combinations of scripts should be considered successful."

//

Understanding AndNode and OrNode Classes - Line by Line Explanation

Let me break down both classes in simple terms and then walk you through an example.

AndNode Class Explanation

```python
class AndNode(LogicalOperatorNode):
    """Node representing an AND operator"""
    def __init__(self, circuit_breaking=True, children=None):
        super().__init__("&&" if circuit_breaking else "&", children)
        self.circuit_breaking = circuit_breaking
```

1. `class AndNode(LogicalOperatorNode):` - Creates a specialized class for AND operations that inherits from LogicalOperatorNode
2. `def __init__(self, circuit_breaking=True, children=None):` - Constructor that takes:
   - `circuit_breaking`: A flag to decide whether to stop evaluation early when possible
   - `children`: Optional list of child nodes (the operands in the AND expression)
3. `super().__init__("&&" if circuit_breaking else "&", children)` - Calls parent constructor with:
   - Operator symbol: `"&&"` for circuit-breaking AND, or `"&"` for non-circuit-breaking AND
   - The provided children list
4. `self.circuit_breaking = circuit_breaking` - Stores the circuit-breaking flag for later use

```python
def evaluate(self, executor_func, verify_hash=False, script_hashes=None):
    """Evaluate AND node with or without circuit breaking"""
    result = True
```

5. `def evaluate(self, executor_func, verify_hash=False, script_hashes=None):` - Method to evaluate the AND expression
6. `result = True` - Starts with assuming the result is True (this is how AND logic works - true until proven false)

```python
for child in self.children:
    child_result = child.evaluate(executor_func, verify_hash, script_hashes)

    if not child_result:
        result = False
        # Circuit breaking: if any child is false, stop evaluation
        if self.circuit_breaking:
            print(f"Circuit breaking AND: stopping at first FALSE result")
            break
```

7. `for child in self.children:` - Loops through each child node (script or operator)
8. `child_result = child.evaluate(executor_func, verify_hash, script_hashes)` - Evaluates the current child
9. `if not child_result:` - Checks if this child returned False
10. `result = False` - If any child is False, the entire AND is False
11. `if self.circuit_breaking:` - If we're using circuit-breaking mode
12. `print(f"Circuit breaking AND: stopping at first FALSE result")` - Prints debug message
13. `break` - Stops processing further children (since AND is already False)

```python
self.result = result
return result
```

14. `self.result = result` - Saves the final result in the node for later reference
15. `return result` - Returns the Boolean result of the AND operation

OrNode Class Explanation

```python
class OrNode(LogicalOperatorNode):
```

```
    """Node representing an OR operator"""
    def __init__(self, circuit_breaking=True, children=None):
        super().__init__("||" if circuit_breaking else "|", children)
        self.circuit_breaking = circuit_breaking
```

1. `class OrNode(LogicalOperatorNode):` - Creates a specialized class for OR operations
2. `def __init__(self, circuit_breaking=True, children=None):` - Constructor similar to AndNode
3. `super().__init__("||" if circuit_breaking else "|", children)` - Passes operator symbol (`"||"` or `"|"`) to parent
4. `self.circuit_breaking = circuit_breaking` - Stores the circuit-breaking flag

```python
def evaluate(self, executor_func, verify_hash=False, script_hashes=None):
    """Evaluate OR node with or without circuit breaking"""
    result = False
```

5. `def evaluate(self, executor_func, verify_hash=False, script_hashes=None):` - Method to evaluate OR expression
6. `result = False` - Starts assuming the result is False (OR logic - false until proven true)

```python
for child in self.children:
    child_result = child.evaluate(executor_func, verify_hash, script_hashes)

    if child_result:
        result = True
        # Circuit breaking: if any child is true, stop evaluation
        if self.circuit_breaking:
            print(f"Circuit breaking OR: stopping at first TRUE result")
            break
```

7. `for child in self.children:` - Loops through each child node
8. `child_result = child.evaluate(executor_func, verify_hash, script_hashes)` - Evaluates the current child
9. `if child_result:` - Checks if this child returned True
10. `result = True` - If any child is True, the entire OR is True
11. `if self.circuit_breaking:` - If using circuit-breaking mode
12. `print(f"Circuit breaking OR: stopping at first TRUE result")` - Prints debug message
13. `break` - Stops processing further children (since OR is already True)

```python
```

```
self.result = result
return result
```

14. `self.result = result` - Saves the final result in the node
15. `return result` - Returns the Boolean result of the OR operation

Example Walkthrough: `|| [ && [ (A), (B) ], !(C) ]`

This expression means: "Either (A AND B) is true, OR (NOT C) is true"

Let's trace through the evaluation with some example values:

Scenario: A succeeds, B fails, C succeeds

1. **Root OR Node** starts evaluation with `result = False`
2. **First Child: AND Node** starts evaluation with `result = True`
   - **Evaluates Script A**: Returns `True` (script A succeeded)
   - **Evaluates Script B**: Returns `False` (script B failed)
   - Since B failed, AND sets `result = False`
   - Circuit breaking triggers: "Circuit breaking AND: stopping at first FALSE result"
   - AND node returns `False`
3. **Back to OR Node**: First child returned `False`
4. **Second Child: NOT Node**
   - **Evaluates Script C**: Returns `True` (script C succeeded)
   - NOT inverts this: `not True = False`
   - Print: "NOT operator: inverting True to False"
   - NOT node returns `False`
5. **Back to OR Node**: Second child also returned `False`
6. **OR Node Final Result**: Both children returned `False`, so OR result stays `False`

Scenario: A succeeds, B succeeds, C not evaluated

1. **Root OR Node** starts evaluation with `result = False`
2. **First Child: AND Node** starts evaluation with `result = True`
   - **Evaluates Script A**: Returns `True` (script A succeeded)
   - **Evaluates Script B**: Returns `True` (script B succeeded)
   - AND node returns `True` (both children successful)
3. **Back to OR Node**: First child returned `True`
4. **OR Node Sets**: `result = True`
5. **Circuit Breaking Triggers**: "Circuit breaking OR: stopping at first TRUE result"
6. **OR Node Final Result**: `True` (C is not evaluated at all)

Scenario: A fails, B not evaluated, C fails

1. **Root OR Node** starts evaluation with `result = False`
2. **First Child: AND Node** starts evaluation with `result = True`
   - **Evaluates Script A**: Returns `False` (script A failed)
   - Circuit breaking triggers: "Circuit breaking AND: stopping at first FALSE result"
   - AND node returns `False` (B is not evaluated)
3. **Back to OR Node**: First child returned `False`
4. **Second Child: NOT Node**
   - **Evaluates Script C**: Returns `False` (script C failed)
   - NOT inverts this: `not False = True`
   - Print: "NOT operator: inverting False to True"
   - NOT node returns `True`
5. **Back to OR Node**: Second child returned `True`
6. **OR Node Sets**: `result = True`
7. **OR Node Final Result**: `True`

Key Points for Your Manager

1. **Circuit Breaking Optimization**: These classes implement "short-circuit evaluation" which is a common optimization in programming languages:
   - For AND operations: If any operand is false, the result must be false, so we can stop evaluating
   - For OR operations: If any operand is true, the result must be true, so we can stop evaluating

2. **Tree-Based Evaluation**: The system builds a tree of operations where:
   - Leaf nodes are scripts to run
   - Internal nodes are logical operators (AND, OR, NOT)
   - Evaluation happens depth-first, with results flowing up the tree

3. **Efficiency**: The circuit breaking ensures we don't run scripts unnecessarily, saving time and resources.

4. **Flexibility**: This design allows for arbitrarily complex logical expressions to control script execution flow.

This approach gives you powerful control over which scripts run based on the success or failure of other scripts, with efficient evaluation that stops as soon as the final result is determined.

//

Understanding the Core Node Classes

Let me explain these foundational classes in a simple way, showing how they connect with `OrNode`, `AndNode`, and `NotNode`.

1. Node Class

```python
class Node:
    """Base class for all syntax tree nodes"""
    def __init__(self):
        self.result = None

    def evaluate(self, executor_func, verify_hash=False, script_hashes=None):
        """To be implemented by subclasses"""
        raise NotImplementedError
```

Simple Explanation:
- This is the abstract base class that all other nodes inherit from
- It's like a blueprint that defines what every node must be able to do
- It has a `result` property to store the evaluation result
- The `evaluate` method is a placeholder that must be implemented by any child class

Purpose: Creates a common interface so that all nodes can be treated similarly when building and evaluating the tree.

2. ScriptNode Class

```python
class ScriptNode(Node):
    """Node representing a script execution"""
    def __init__(self, name, args=None):
        super().__init__()
        self.name = name
        self.args = args or []

    def evaluate(self, executor_func, verify_hash=False, script_hashes=None):
        """Execute the script and return result"""
        expected_hash = script_hashes.get(self.name) if verify_hash and script_hashes else None
        self.result = executor_func(self.name, self.args, verify_hash, expected_hash)

        # In logical context, 0 (success) = True, 1 (failure) = False
        logical_result = (self.result == 0)

        # Write the result code to stderr for individual script
        sys.stderr.write(f"{self.result}\n")
```

```
        return logical_result
```

Simple Explanation:
- This represents an actual script to execute (like script A, B, or C)
- It stores:
  - `name`: The name of the script to run (e.g., "A")
  - `args`: Optional arguments to pass to the script (e.g., ["hello", "world"])
- Its `evaluate` method:
  1. Gets the expected hash for this script (if available)
  2. Calls the executor function to run the script
  3. Converts the result code (0=success, 1=failure) to a boolean (0=True, 1=False)
  4. Outputs the raw result to stderr
  5. Returns the boolean result

Purpose: These are the "leaf nodes" of the tree - they do the actual work of running scripts.

3. LogicalOperatorNode Class

```python
class LogicalOperatorNode(Node):
    """Node representing a logical operator with children"""
    def __init__(self, operator, children=None):
        super().__init__()
        self.operator = operator
        self.children = children or []

    def add_child(self, child):
        self.children.append(child)

    def __str__(self):
        children_str = ", ".join(str(child) for child in self.children)
        return f"{self.operator} [ {children_str} ]"
```

Simple Explanation:
- This is a base class for all logical operators (AND, OR)
- It stores:
  - `operator`: The symbol for the operator (like "&&" or "||")
  - `children`: A list of child nodes to operate on
- It has an `add_child` method to build the tree
- Its `__str__` method creates a string representation like "&& [(A), (B)]"
- The actual evaluation logic is implemented in subclasses (AndNode, OrNode)

Purpose: Provides common functionality for all logical operators that work with multiple child nodes.

How These Classes Connect Together

These three base classes form the foundation of the tree structure:

1. **Node** - The abstract base class everything inherits from
2. **ScriptNode** - Leaf nodes that run individual scripts
3. **LogicalOperatorNode** - Branch nodes that combine results of other nodes

The specialized nodes then inherit from these base classes:

- **AndNode** inherits from LogicalOperatorNode
- **OrNode** inherits from LogicalOperatorNode
- **NotNode** inherits directly from Node (since it has only one child, not multiple)

Example with `|| [ && [ (A), (B) ], !(C) ]`

Let's see how these classes work together in our example:

```
         OR (LogicalOperatorNode → OrNode)
        /    \
AND (LogicalOperatorNode → AndNode)     NOT (Node → NotNode)
    /  \                          |
   A    B                         C
(ScriptNode) (ScriptNode)          (ScriptNode)
```

In this tree:

1. The root is an `OrNode` (which is a type of `LogicalOperatorNode`)
   - It has two children: an `AndNode` and a `NotNode`

2. The `AndNode` (which is a type of `LogicalOperatorNode`)
   - It has two children: script A and script B (both `ScriptNode` instances)

3. The `NotNode` (which inherits directly from `Node`)
   - It has one child: script C (a `ScriptNode` instance)

When evaluation happens:

1. The `OrNode` needs to evaluate its children

2. It first asks the `AndNode` to evaluate
3. The `AndNode` evaluates its `ScriptNode` children (A and B)
   - Each `ScriptNode` runs its actual script using the executor function
   - Each returns True/False based on script success/failure
4. If needed, the `OrNode` then asks the `NotNode` to evaluate
5. The `NotNode` evaluates its `ScriptNode` child (C)
   - Script C runs using the executor function
   - NotNode inverts the result

Real-World Analogy

Think of it like a company structure:

- **Node** is like the basic "Employee" role - everyone has certain responsibilities
- **ScriptNode** is like the "Worker" who actually performs tasks
- **LogicalOperatorNode** is like a "Manager" who coordinates multiple employees
- **AndNode** is a "Strict Manager" who requires all tasks to succeed
- **OrNode** is a "Lenient Manager" who is satisfied if any task succeeds
- **NotNode** is a "Contrary Supervisor" who reports the opposite of what happened

Key Points for Your Manager

1. **Object-Oriented Design**: The code uses inheritance to create a flexible, extensible system
2. **Tree Structure**: The nodes form a tree where evaluation flows from leaves (scripts) up to the root
3. **Separation of Concerns**:
   - `ScriptNode` handles script execution
   - `LogicalOperatorNode` and subclasses handle combining results
   - `NotNode` handles inversion of results
4. **Polymorphism**: All nodes have an `evaluate` method, so they can be used interchangeably

This design makes it easy to:
- Add new types of nodes in the future
- Build arbitrarily complex logical expressions
- Evaluate expressions efficiently with circuit breaking

//

When OR Adds Its Children During Parsing

Let me explain in a simple way when and how an OR node adds its children during the parsing process.

The Step-by-Step Process

When parsing an expression like `|| [ && [ (A), (B) ], !(C) ]`:

1. **First, the OR node is created**:
   - The parser sees `||` at the beginning
   - It creates a new `OrNode` object
   - But at this point, the OR node has NO children yet

2. **The parser then looks for the OR node's children**:
   - It calls `_parse_operator_children(expr, pos+2, or_node)`
   - This function is responsible for finding and adding all children to the OR node

3. **For each child of the OR node**:
   - The parser recursively calls `_parse_expression`
   - This means it fully builds each child (and any descendants) before adding it to the OR node

4. **For the first child (AND node)**:
   - The parser sees `&&`
   - Creates an `AndNode`
   - Recursively finds and adds the AND node's children (A and B)
   - **Only after the AND node is completely built** with its own children
   - The AND node is added to the OR node with `or_node.add_child(and_node)`

5. **For the second child (NOT node)**:
   - The parser sees `!`
   - Creates a `NotNode`
   - Recursively finds and sets the NOT node's child (C)
   - **Only after the NOT node is completely built** with its child
   - The NOT node is added to the OR node with `or_node.add_child(not_node)`

Simple Analogy

Think of it like building a family tree:

1. First, you identify a grandparent (OR node)
2. To add their children, you must:
   - Fully identify each child AND their own children (grandchildren)
   - Only after a complete "family branch" is built do you connect it to the grandparent

Looking at the Code

Here's the key part in `_parse_operator_children`:

```python
# Parse children until closing bracket
while pos < len(expr) and expr[pos] != "]":
    # Skip whitespace and commas
    while pos < len(expr) and (expr[pos].isspace() or expr[pos] == ","):
        pos += 1

    # Check if we're at the end of children
    if pos < len(expr) and expr[pos] != "]":
        child, pos = _parse_expression(expr, pos)  # This builds the entire child subtree
        if child:
            node.add_child(child)  # Only after child is fully built, it's added to the parent
```

In Summary

The OR node adds its children:
- **After** each child is fully constructed with its own descendants
- **During** the parsing process (not at the end of all parsing)
- **One at a time** in the order they appear in the expression

So for `|| [ && [ (A), (B) ], !(C) ]`:
1. First, the complete AND node with its A and B children is built
2. Then the complete AND node is added to the OR node
3. Next, the complete NOT node with its C child is built
4. Finally, the complete NOT node is added to the OR node

This "depth-first" approach ensures each node has its complete subtree before being added to its parent.