NAME

Chain-of-Command-Frick (CoCF) – A Brainfuck-inspired encoding schema for multi-agent network topologies

SYNOPSIS

A string of symbols (`>`, `<`, `+`, `-`, `[`, `]`, `.`, `,`, `@`, `*`) encodes a "chain-of-command" communication structure among agents. Execution modifies hierarchical links, priorities, and message passing rules.

DESCRIPTION

Symbol	Meaning	Functionality
`>`	Shift communication focus to subordinate	The current agent communicates with its immediate subordinate in the CoC.
`<`	Shift communication focus to superior	Escalate communication or feedback to a superior in the CoC.
`+`	Strengthen connection	Increase communication intensity or priority for the current focus node.
`-`	Weaken connection	Reduce communication intensity or deprioritize the current focus node.
`[ ]`	Conditional delegation	If a condition is met, delegate tasks or propagate information to subordinates.
`.`	Transmit message	Pass task or data to the currently focused node.
`,`	Receive input	Retrieve results or data from the currently focused node.
`@`	Broadcast to all subordinates	Send a task or message to all directly connected subordinate nodes.
`*`	Lateral information sharing	Share information laterally to siblings in the same hierarchical level.

IMPLEMENTATION

```python
import sys

class Node:
    def __init__(self, node_id, parent=None):
        self.id = node_id
        self.parent = parent
        self.children = []
        self.broadcast = False
        self.messages_sent = 0
        self.messages_received = 0

class CoCFInterpreter:
    def __init__(self, code):
        self.code = code
        self.nodes = [Node(0, None)]
        self.edges = {}
        self.focus = 0
        self.pointer = 0

    def _create_child(self, parent_id):
        new_id = len(self.nodes)
        child = Node(new_id, parent_id)
        self.nodes.append(child)
        self.nodes[parent_id].children.append(new_id)
        self.edges[(parent_id, new_id)] = {"weight": 1}
        return new_id

    def _create_sibling(self):
        current = self.nodes[self.focus]
        if current.parent is None:
            return self._create_child(0)
        parent_id = current.parent
        siblings = self.nodes[parent_id].children
        idx = siblings.index(self.focus)
        if idx < len(siblings)-1:
            return siblings[idx+1]
        else:
            return self._create_child(parent_id)

    def _modify_edge_weight(self, delta):
        current = self.nodes[self.focus]
        if current.parent is not None:
            e = (current.parent, self.focus)
            new_weight = self.edges[e]["weight"] + delta
            if new_weight < 1:
                new_weight = 1
            self.edges[e]["weight"] = new_weight

    def execute(self):
        while self.pointer < len(self.code):
            c = self.code[self.pointer]
            if c == '>':
                cur = self.nodes[self.focus]
                if cur.children:
                    self.focus = cur.children[0]
                else:
                    self.focus = self._create_child(self.focus)

            elif c == '<':
                cur = self.nodes[self.focus]
                if cur.parent is not None:
                    self.focus = cur.parent

            elif c == '+':
                self._modify_edge_weight(+1)

            elif c == '-':
                self._modify_edge_weight(-1)

            elif c == '.':
                self.nodes[self.focus].messages_sent += 1

            elif c == ',':
                self.nodes[self.focus].messages_received += 1

            elif c == '@':
                self.nodes[self.focus].broadcast = True

            elif c == '*':
                self.focus = self._create_sibling()

            elif c == '[' or c == ']':
                pass

            self.pointer += 1

    def to_dot(self):
        lines = ["digraph G {"]
        for n in self.nodes:
            label_parts = [f"N{n.id}"]
            if n.broadcast:
                label_parts.append("B")
            if n.messages_sent > 0:
                label_parts.append(f"S{n.messages_sent}")
            if n.messages_received > 0:
                label_parts.append(f"R{n.messages_received}")
            label = "\\n".join(label_parts)
            lines.append(f'    N{n.id} [label="{label}"];')
        for (p, c), attrs in self.edges.items():
            w = attrs["weight"]
```

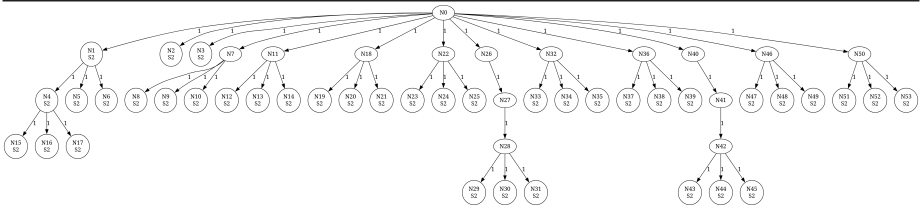
```
                lines.append(f'    N{p} -> N{c} [label="{w}"];')
            lines.append("}")
            return "\n".join(lines)

def main():
    code = sys.stdin.read().strip()
    interpreter = CoCFInterpreter(code)
    interpreter.execute()
    print(interpreter.to_dot())

if __name__ == "__main__":
    main()
```

EXAMPLES

```
>..*..*..<>>..*..*..<<*>..*..*..<<*>..*..*..<<>>>..*..*..<<<<
*>..*..*..<<<<*>..*..*..<<<<*>>>..*..*..<<<<*>..*..*..<<<<
*>..*..*..<<<<*>>>..*..*..<<<<*>..*..*..<<<<*>..*..*..<<<<
```



LIMITATIONS

- Conditional loops (`[ ]`) are not yet implemented.
- The current implementation focuses on hierarchical navigation, task delegation, and link adjustments.
- Future iterations may support more complex logic and conditions.