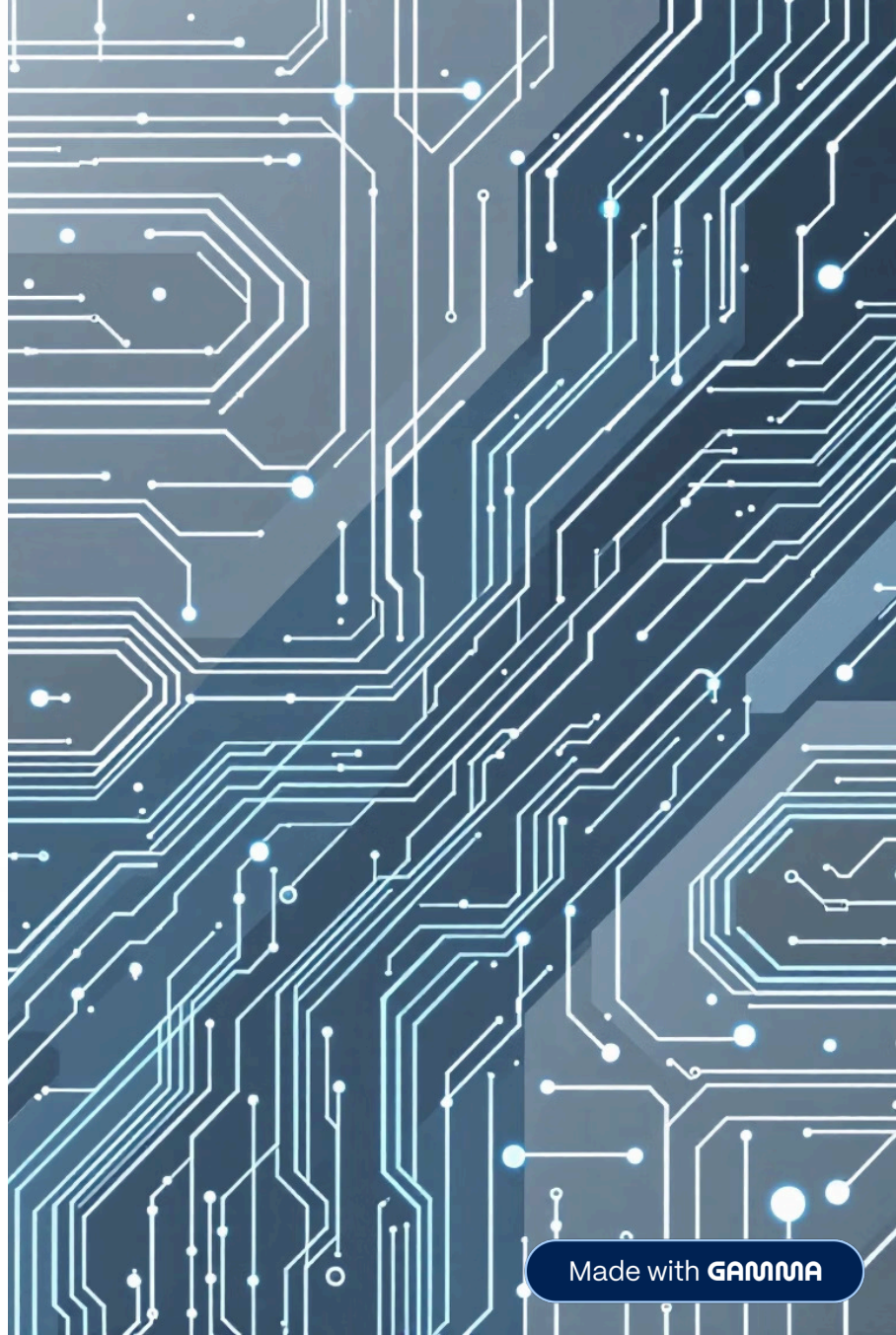
A Turing Machine-Inspired Approach to Code Formatting

Design and Implementation of a Multi-Pass C-Like Code Beautifier

Kushagra Singh, Shrut Jain, Rishabh Jain, Harsh Ramkete



The Challenge of Code Readability

In software engineering, code is read far more often than it is written. Inconsistent formatting (indentation, spacing, brace style) increases cognitive load and hinders collaboration.

Manual formatting is tedious and error-prone. Automated formatters (beautifiers) are essential tools for maintaining consistency and improving code quality across large projects and teams.

The Problem: Most advanced formatters rely on complex Abstract Syntax Trees (ASTs) for parsing and restructuring code. This approach, while powerful, can be computationally intensive and difficult to implement.

Our Question: Can we build a robust and functional formatter using a simpler, more fundamental computational model, such as one inspired by a Turing Machine?

Before (Messy Code)

```
int main(){int a=3;int b= 4; if(a<b) b = b + a; for(int i=0;i<10;i++){printf(\"%d\",i);}return 0;}
```

After (Clean Code)

```
int main() {
 int a = 3;
 int b = 4;
 if (a < b) {
 b = b + a;
 }
 for (int i = 0; i < 10; i++) {
 printf("%d", i);
 }
 return 0;
}
```

Project Objectives

1

TM-Inspired Design

To design a code formatter based on the Turing Machine (TM) computational model, specifically utilizing a "tape" and a "head" metaphor for code manipulation.

2

Multi-Pass System

To implement the formatting logic as a multi-pass system, where each distinct pass incrementally refines and beautifies the code.

3

C-Like Syntax Handling

To handle key C-like syntax features, including operator and brace spacing, newline insertion after semicolons, and correct block-level indentation.

4

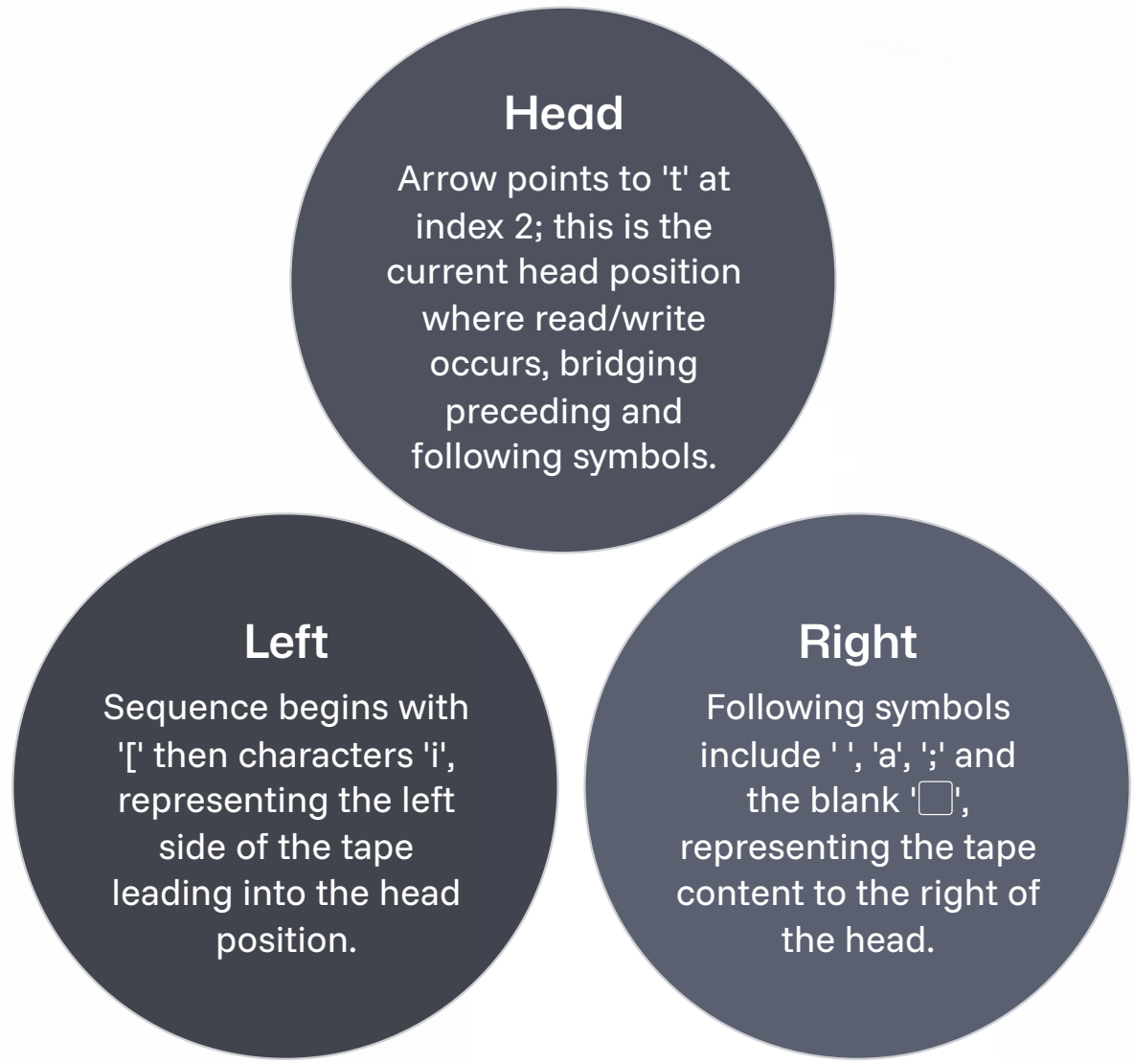
Complex Case Resolution

To solve complex formatting cases such as protecting `for` loop headers from premature newlines and auto-inserting braces (`{}`) for single-statement bodies to improve readability and prevent common errors.

Methodology: The Turing Machine-Inspired Model

Our approach simulates the fundamental storage model of a Turing Machine, focusing on its mechanism of sequential data access and manipulation, rather than replicating a formal state machine.

- **The Tape:** The entire source code is treated as a linear sequence of characters. It is loaded into a single mutable character list (`List[str]`), which serves as our "tape."
- **The Head:** An integer index (`self.head`) precisely represents the TM's read/write head, pointing to one character on the tape at a time. This head can move left or right, reading and modifying characters.
- **The "Blank" Symbol:** A special character (␣) is used to explicitly represent the conceptual end of the tape, ensuring boundary conditions are handled gracefully during operations.



Implementation: The Core Primitives

All tape manipulation operations are abstracted into a set of five fundamental primitives, directly mimicking the core actions of a Turing Machine. This design choice significantly simplifies the implementation of all higher-level formatting logic.

```
# --- Core TM Primitives ---
def read(self) -> str:
 # Returns character at self.head
 return self.tape[self.head]

def write(self, ch: str):
 # Overwrites character at self.head
 self.tape[self.head] = ch

def move_right(self):
 self.head += 1

def move_left(self):
 self.head = max(0, self.head - 1)

def insert(self, ch: str):
 # Inserts character at self.head
 self.tape.insert(self.head, ch)

def delete(self):
 # Deletes character at self.head
 del self.tape[self.head]
```

Architecture: A Multi-Pass Formatter

Rather than attempting to perform all formatting within a single, monolithic pass, our system employs a sequence of simple, specialized "sub-machines." Each pass scans the entire tape to perform one specific task, ensuring logic remains clean, deterministic, and easier to debug.

```
def format(self):
 # 1. Clean and normalize the tape
 self.pass_normalize_whitespace()

 # 2. Handle special syntax cases
 self.pass_protect_for_headers()
 self.pass_space_around_operators()
 self.pass_autobrace_single_statements()

 # 3. Apply structural formatting rules
 self.pass_semicolon_newlines()
 self.pass_restore_placeholders()
 self.pass_braces_newlines()

 # 4. Final cleanup and indentation
 self.pass_trim_blank_lines()
 self.pass_indentation()
```

Key Feature: Handling `for` Loop Semicolons

The Problem: A naive "newline after semicolon" pass would inadvertently break the structure of `for` loops, transforming functional code into syntactically incorrect or unreadable blocks.

```
for(int i=0; i<10; i++) would become:
for(int i=0;

i<10;

i++)
```

The Solution (A 3-Pass Process):

1

1. Protect Headers

`pass_protect_for_headers`: Scans for `for(...)` constructs and temporarily replaces any semicolons within the parentheses with a special `PLACEHOLDER` character (e.g., `\x07`).

2

2. Newline Pass

`pass_semicolon_newlines`: Executes normally, adding newlines after all real semicolons in the code while completely ignoring the placeholder characters.

3

3. Restore Placeholders

`pass_restore_placeholders`: Scans the tape one final time, converting all `PLACEHOLDER` characters back into their original semicolons, thus preserving the `for` loop structure.

Key Feature: Automatic Brace Insertion

The Problem: Single-statement bodies (e.g., `if(a) b;`) for conditional or loop statements are a common source of bugs and can decrease code clarity. Omitting braces often leads to misinterpretations or unintended execution flows when adding subsequent lines of code.

The Solution: The `pass_autobrace_single_statements` pass systematically detects these cases and inserts braces to ensure explicit block definition:

- It identifies keywords such as `if`, `for`, `while`, and `else`.
- It then scans past the header (e.g., `(...)`).
- It reads the next non-whitespace character after the header.
- If this character is not an opening brace (`{`), the machine performs the following actions:
  - Inserts an opening brace `{` at the current head position.
  - Moves forward, carefully balancing parentheses `()`, to locate the terminating semicolon `;` of the single statement.
  - Inserts a closing brace `}` immediately after this semicolon.

A crucial special case prevents this logic from running between an `else` keyword and a subsequent `if` statement (`else if`) to maintain proper conditional chaining.

Demonstration

Before

```
int main(){int a=3;int b= 4; if(a<b) b = b + a; for(int i=0;i<10;i++){printf(\"%d\",i);}return 0;}
```

After

```
int main() {
    int a = 3;
    int b = 4;
    if (a < b) {
        b = b + a;
    }
    for (int i = 0; i < 10; i++) {
        printf("%d", i);
    }
    return 0;
}
```

Note the automatic insertion of spaces around operators, consistent newlines, correct indentation levels, and crucially, the automatic insertion of braces around the single-statement `if` body (`b = b + a;`).

Challenges & Limitations

Performance

This model is primarily academic. Using Python's native list insert/delete operations results in `O(n)` complexity for each edit. This makes the total runtime `O(n  m)` (where `n` is code length and `m` is the total number of passes/edits), which is significantly slower than production-grade formatting tools designed for efficiency.

No True Parser

The formatter is "parser-less" in the traditional sense; it operates without a deep understanding of code context. It cannot reliably distinguish semicolons inside string literals (e.g., `"hello;"`) or comments (e.g., `// my; code`). This limitation means complex C/C++ macros or template metaprogramming constructs would easily break its logic.

State Management

The "state" is managed through procedural Python code and implicitly by the head's position. A more formal Turing Machine would require an explicit state transition table, which would become extremely large and complex to define for even a modest set of formatting rules, making it impractical for this kind of application.