What this tool is actually building

This tool doesn't build a CFG of Python statements. It builds a **paper-matching "keyword CFG"**:

Nodes (examples)

- `start`, `end`
- `if1`, `ifexp1`
- `elif2`, `elifexp2`
- `else4`
- `for5`, `while6`
- nested variants: `ifsp7`, `ifspexp7`, `forsp8`, `whilesp9`, `elsesp10`

Edges (paper-style, for top-level if/elif/else chains)

- `ifN -> ifexpN`
- `ifN -> (next branch keyword: elifM / elseK / end)`
- `ifexpN -> end`
- `elifN -> elifexpN`
- `elifN -> (next branch keyword: elifM / elseK / end)`
- `elifexpN -> end`
- `elseN -> end`

Loops are handled with your older heuristic logic (self-loops + fallthrough-ish edges).

---

1) `cfg_builder.py` — the core algorithm

This file has **two phases**:

Phase A: `build_listM(source)` → "ListM"

This reads the source *line-by-line* and extracts only leading keywords in:

```
KEYWORDS = ["if", "elif", "else", "for", "while"]
```

How it decides nesting

It uses indentation:

- `indent_level(line)` counts leading spaces (tabs treated as 4 spaces)
- a `stack[(indent, label)]` tracks currently-open keyword blocks

- before handling a line, it pops stack frames when indentation decreases:
  `while stack and lvl <= stack[-1][0]: stack.pop()`
- `nested = bool(stack)` → if nested, it uses `sp` labels

The "one global counter" rule

`count` increments **for every keyword occurrence** (if/elif/else/for/while).
That's what makes numbering match the paper example ordering.

Tokens it emits

- for `if` / `elif`: emits **two tokens**: `ifN` and `ifexpN` (or `ifspN` / `ifspexpN`)
- for `else`: emits only `elseN` (or `elsespN`)
- for `for` / `while`: emits only `forN` / `whileN` (or `forspN` / `whilespN`)

Then it inserts:

- `"start"` at front
- `"end"` at end

So `ListM` is the raw "token stream".

Phase B: `generate_cfg(listM)` → nodes + edges

Outputs:

- `nodes`: unique tokens in order
- `edges`: List2 (pairs)
- `list3`: bookkeeping list (looks like "visited CFG tokens")
- `freq`: counts of tokens from `list3`

Paper-style special handling: top-level if/elif/else chains

This is the biggest "match the paper" logic.

The helper matchers:

- `_is_plain_if("if12")`
- `_is_plain_elif("elif3")`
- `_is_plain_else("else4")`
- `_is_plain_exp("ifexp1" / "elifexp2")`
- `_is_branch_kw`: elif/else/end

Then the algorithm scans `listM`:

Rule 1: exp nodes don't fall through

If current token is `ifexpN` or `elifexpN`, it just skips normal fallthrough.
(exp edges are added when the parent `ifN/elifN` is processed.)

Rule 2: for `ifN` or `elifN` followed by exp node

Example: `if1, ifexp1`

It adds:

- `if1 -> ifexp1`
- `if1 -> next_branch` where next_branch is found by scanning forward for the next `elifK`, `elseK`, or `end`
- `ifexp1 -> end`

Same for `elifN`.

Rule 3: for `elseN`

- `elseN -> end`

That's it.

Loop logic (legacy heuristic)

This part is *not* paper-precise; it's heuristic and a bit inconsistent:

- `forN` creates:
  - `forN -> forN` self-loop sometimes
  - `forN -> first non-nested token ahead`
  - `forN -> next token` if different from that first non-nested token
- `whileN` similar
- `forspN` / `whilespN` do extra weird edges involving `"f"` (a special token that isn't guaranteed to be in nodes!)
- nested ifsp tokens mostly default to "fallthrough to next"

Start edge

Always adds:

- `start -> listM[1]` (first token after start)

De-duplication

Edges are de-duplicated while preserving order.

`build_adj_from_edges(edges)`

Just makes adjacency dict `{src: [dst1, dst2, ...]}`.

---

2) `gui.py` — UI wrapper around the token CFG

This is a Tkinter app that does:

1. User pastes code
2. `build_listM(src)` and `generate_cfg(listM)`
3. Shows:
   - ListM
   - nodes
   - edges
4. Creates DOT text from nodes/edges
5. Uses `graphviz` Python package to render DOT → SVG
6. Opens SVG in browser (Tkinter can't display SVG)

Important details

- It forces PATH to include common Graphviz install dirs on macOS: `:/opt/homebrew/bin:/usr/local/bin`
- `edges_to_dot()` uses very simple DOT (no labels on edges, no styling besides `rankdir=TB`)

So your CFG graph is purely structural (node names only).

---

3) `main.py` — CLI example runner

Runs the same pipeline:

- `listM = build_listM(sample_program)`
- `nodes, edges, list3, freq = generate_cfg(listM)`
- prints everything
- builds adjacency
- computes paths from `start` to `end` using `find_all_paths`

---

4) `paths_finder.py` — path enumeration with cycle handling

This is a translation of your pseudocode. It's *not* a standard DFS.

Core behavior:

- Builds `path = path + [source]`
- If `source == dest`, returns `[path]`
- For each neighbor `node`:
    - If `node` already in path (cycle detected):
        - it inspects `d[node]` and:
            - if sees self-loop (`i == node`) → break
            - if sees `end` → it *forces* `[... node, end]` as a path and appends it
    - Else:
        - recurse normally

So it's explicitly designed to handle cycles caused by loop self-edges.

How this differs from python-graphs / staticfg (conceptually)

This method:

- works off **text + indentation**
- ignores non-keyword control flow (return, break, continue, try/except, raise, function calls)
- models only a *subset* of constructs
- produces a CFG that is best understood as a **paper diagram reproduction**, not an executable-accurate CFG

staticfg / python-graphs:

- AST-based
- represent real control flow (staticfg partially; python-graphs much more fully)

Key pitfalls / bugs to be aware of in this code

1. **`"f"` token**
   Some loop branches add edges to `"f"`:

```
edges.append((cur, "f"))
```

But `"f"` is never guaranteed to exist in `nodes` (unless it appeared in ListM, which it won't). Graphviz will still draw it as an implicit node, but your node list / complexity counts get inconsistent.

2. **Exp nodes always go to `end`**
   Paper-matching, yes — but semantically it collapses all "then-body" behavior into a single `end`.
   So paths/complexity aren't a real cyclomatic complexity of the program.
3. **Indent stack pop condition**
   `while stack and lvl <= stack[-1][0]: pop()`
   This treats same-indent as "ended" the previous block, which can be okay for Python blocks, but it's a blunt heuristic and can misclassify in some layouts.
4. **No parsing**
   A line like:

```
if_condition = 3
```

would be ignored (good), but something like:

```
if(x):  # non-python style or comments weirdness
```

or multiline conditions can throw the heuristic off. It's not using `ast`.

One sentence summary

This "Token Heuristic Method" builds a **keyword-token CFG** (ListM → edges) that's optimized to **match a paper's diagram style** for if/elif/else chains, while loops are handled by older heuristic self-loop logic; the GUI just visualizes the resulting token graph and the path finder enumerates start→end paths with special cycle handling.