Phase 02

Code Summarization and Generation

Laiba Akram

42943

BSCS-7A

Theory Of programming Laguage

Phase 2 – Methodology Document

1. Overview

The goal of Phase 2 is to convert raw source code into structured, PL-aware data that can be analyzed quantitatively. Instead of treating code as plain text, we represent it using Abstract Syntax Trees (ASTs), control-flow properties, and language-level features (typing discipline, memory safety, concurrency constructs).

This phase produces:

1. A cleaned, labeled dataset of code snippets with language metadata.
2. A feature matrix containing syntax- and PL-theoretic metrics for each snippet.

These features will later be used to test the hypothesis from Phase 1 about how PL features and syntax complexity relate to LLM performance in code generation and summarization.

2. Data Collection

2.1 Data Sources

We will use a combination of public benchmarks and open-source repositories:

- CodeSearchNet / CodeXGLUE-style datasets for:
    - Python
    - Java
    - JavaScript
- GitHub repositories (via API or manual download) for:
    - Rust (to ensure we include a strongly memory-safe, statically typed language).

All datasets will be limited to open-source, permissively licensed projects.

2.2 Sampling Strategy

- Unit of analysis: *Function- or method-level snippets* (not entire files).
- Target size:
    - ~3,000–5,000 functions per language after cleaning (adjustable depending on parsing success).

- Selection criteria:
    - Functions between 5 and 200 lines (excluding comments and blank lines).
    - Exclude:
        - Auto-generated code (files with “Generated by”, “do not edit”, etc.).
        - Vendor / dependency folders (node_modules, target, build, etc.).
        - Test files (heuristics like filenames including test, spec, *_test.*).

Each function will be stored with a globally unique function_id and metadata:

- repo_name, file_path, language, function_name, start_line, end_line.

3. Preprocessing

Preprocessing transforms raw files into normalized code snippets.

3.1 Parsing & Extraction

For each repository:

1. Detect language via file extension (e.g., .py, .java, .rs, .js) or using a language detection tool.
2. Use a language-appropriate parser (e.g., Tree-sitter / ANTLR / official compiler front-end) to:
    - Parse the file.
    - Extract function/method definitions as standalone code units.
3. Store each function’s:
    - Raw code.
    - AST (in a serializable format, e.g., JSON, Protobuf, or custom node structure).

3.2 Cleaning

For each function:

- Strip:
    - Inline comments and block comments (unless specifically needed for summarization tasks).

    - Trailing whitespace, extra blank lines.
- Normalize:
    - Indentation (e.g., 4 spaces) to avoid formatting differences across projects.
    - Line endings (LF).
- Optionally anonymize:
    - Variable and function names (e.g., rename to var1, var2, func1, etc.) *only if* required to isolate structural complexity from naming. (You can keep original identifiers if you want later semantic analysis.)

4. Feature Engineering (PL-Grounded)

This is the core "TCPL / PL theory" part: we treat code as structure (AST, control-flow, PL features) rather than text.

We define the unit for feature extraction as a single function/method.

4.1 Language-Level Features (Per Function)

For each function, we attach language-level PL descriptors:

- lang ∈ {Python, Java, Rust, JavaScript}
- is_static_typed ∈ {0,1}
    - Python/JavaScript = 0
    - Java/Rust = 1
- is_memory_safe ∈ {0,1}
    - Rust = 1 (strong guarantees)
    - Java ≈ 1 (managed memory, no manual free) — you can encode as 1 or a separate category
    - Python/JavaScript also managed; optionally keep a more nuanced categorical encoding:
        - memory_model ∈ {"manual", "managed", "ownership"}

- **paradigm (one-hot encoded):**
    - Imperative/OOP: Java, C-style features in Rust, Python, JavaScript
    - Functional-friendly / multi-paradigm: Rust, Python, JavaScript
- **concurrency_model (categorical):**
    - “threads + locks”: Java, Rust (std::thread, mutex)
    - “async/await”: Python (asyncio), JavaScript (event loop)
    - You can initially mark at language level and later refine with per-function concurrency features (below).

These features explicitly encode PL theory concepts (typing discipline, memory model, paradigm, concurrency model), not just text statistics.

4.2 Syntax Complexity via AST

Using the AST for each function, compute **structural metrics**:

1. **AST Size & Depth**
    - ast_node_count: Total number of nodes in the AST.
    - ast_depth: Longest path length from root to leaf.
    - ast_leaf_count: Number of leaf nodes.
    - ast_branching_factor_avg:
- avg children per internal node
    - ast_distinct_node_types: Number of unique AST node kinds (e.g., IfStatement, ForLoop, FunctionCall).
2. **Statement & Expression Counts**
    - num_statements
    - num_expressions
    - num_if, num_for, num_while, num_switch, num_try_catch, etc.
    - num_function_calls

3. **Operator & Operand Diversity (syntax-aware)**
    - num_distinct_operators (e.g., arithmetic, logical, comparison, assignment).
    - num_distinct_literals (numeric, string, boolean, etc.).

These directly implement the idea of "syntax complexity" as structure, not tokens.

4.3 Control-Flow and Classic Complexity Measures

From the AST or derived Control Flow Graph (CFG):

1. **Cyclomatic Complexity (McCabe)**
    - cc_mccabe:

$$\mathrm{M} = E - N + 2P$$

where E= edges, N= nodes in CFG, P= number of connected components (usually 1 per function).

    - Equivalent heuristic: count decision points (if, for, while, case, &&, ||, ? :).
2. **Halstead Metrics**
    - n1 = number of distinct operators
    - n2 = number of distinct operands
    - N1 = total operators
    - N2 = total operands
    - From these:
        - halstead_v (volume)
        - halstead_d (difficulty)
        - halstead_e (effort)
    - These are calculated using token categorization derived from the AST/token stream, not just raw text.
3. **CFG Shape Features**
    - num_basic_blocks
    - cfg_branching_factor_avg

- has_loops ∈ {0,1}
- has_exception_handling ∈ {0,1}

Again, all of these are grounded in **compiler theory and program analysis**.

4.4 Concurrency Features

To connect with concurrency-related hypotheses, extract **per-function concurrency indicators** using parsing/AST and, where needed, additional regex filters:

- **Keyword / Construct Counts:**
  - For Python:
    - count_async, count_await, usage of async def, asyncio calls.
  - For Java:
    - count_thread_new (e.g., new Thread, ExecutorService),
    - count_synchronized, count_lock, count_future.
  - For Rust:
    - count_spawn (std::thread::spawn, tokio::spawn),
    - count_mutex, count_arc, count_channel (mpsc, crossbeam_channel).
  - For JavaScript:
    - count_async, count_await,
    - count_promise, count_then, count_setTimeout/setInterval.
- **Binary Flags:**
  - uses_concurrency ∈ {0,1} (if any of the above > 0).
  - concurrency_pattern (categorical: none, threading, async_await, event_loop).

These are **not just regexes over text**; they are supported by AST node types and import/identifier analysis when possible (e.g., confirming that Mutex is actually from std::sync).

4.5 Size & Formatting Controls

To avoid confounding effects of "huge functions vs tiny ones," include control variables:

- loc – Lines of code (excluding comments/blank lines).

- num_parameters
- num_local_variables
- num_return_statements

These will help you normalize or stratify results later.

---

5. Vectorization

Once features are extracted, convert them into **vectors** suitable for clustering or regression.

5.1 Handcrafted Feature Vector

For each function f, define:

$\mathbf{x}_f$
$=$ [ast_node_count,ast_depth,ast_branching_avg,cc_mccabe,halstead_v,halstead_e,num_if,num_loops,uses_conc

- **Numeric features**: kept as floats/ints and later standardized (z-score).
- **Categorical PL features** (language, paradigm, concurrency model) encoded as:
  - One-hot vectors (e.g., [lang_python, lang_java, lang_rust, lang_js]), or
  - Ordinal/categorical encodings if appropriate.

5.2 Embedding-Based Vectorization (Optional but Powerful)

To capture **semantic** information about the code (beyond complexity), you can also compute:

- **Code embeddings** using models like **CodeBERT**, **GraphCodeBERT**, or an LLM's embedding API.
- Denote this embedding as $\mathbf{e}_f \in \mathbb{R}^d$.

You can then either:

- Concatenate: $\mathbf{z}_f = [\mathbf{x}_f \parallel \mathbf{e}_f]$, or
- Use $\mathbf{x}_f$ for PL/complexity analysis and $\mathbf{e}_f$ for clustering / similarity tasks.

In the rubric, emphasize that **core features are AST/CFG-based and PL-theoretic**; embeddings are an additional layer.

---

6. Cleaned Dataset Specification

You will submit **two main artifacts**:

6.1 Tabular Feature Dataset

Format: CSV / Parquet / JSONLines.

Example columns:

- Identification:
  - function_id
  - repo_name
  - file_path
  - language
- PL Features:
  - is_static_typed
  - memory_model
  - paradigm_* (one-hot)
  - concurrency_model
- Syntax / AST Features:
  - loc
  - ast_node_count
  - ast_depth
  - ast_leaf_count
  - ast_branching_factor_avg
  - ast_distinct_node_types
  - num_statements
  - num_expressions
  - num_if, num_for, num_while, num_switch, num_try_catch
- Control-Flow & Metrics:
  - cc_mccabe

    - halstead_n1, halstead_n2, halstead_N1, halstead_N2
    - halstead_v, halstead_d, halstead_e
    - num_basic_blocks
    - cfg_branching_factor_avg
- Concurrency Features:
    - uses_concurrency
    - count_async, count_await, count_thread, count_mutex, count_channel, count_promise, ...
- Optional:
    - embedding (if stored as separate file, refer to function_id).

6.2 Code Snippet Dataset

Format: JSONLines or CSV with text fields.

- function_id
- language
- code_raw (original function code)
- code_clean (after preprocessing)
- Optionally: ast_json (if you want to provide structural representation).

This satisfies the requirement to **"provide the processed data"**.

---

7. Quality Control & Sanity Checks

Before using the dataset in Phase 3:

1. **Parsing success rate**
    - Report % of functions per language where AST was successfully built.
2. **Distribution checks**
    - Histograms of ast_depth, ast_node_count, cc_mccabe per language.
    - Ensure no extreme outliers (e.g., gigantic auto-generated functions).
3. **PL feature validation**

- Spot-check 20–30 functions per language to verify:
  - uses_concurrency is correct when concurrency constructs are present.
  - is_static_typed, memory_model etc. are correctly assigned.

4. **Leakage checks (if used with LLM outputs later)**
   - Ensure training/test splits are on **project level** to avoid the same function appearing in both.

Rubric Alignment (Explicit)

- **"How do you represent code?"**
  → By ASTs, CFG-derived metrics, and PL-level flags, not bag-of-words.
- **"Features extracted based on PL theory?"**
  → Yes: AST node density, depth, branching, cyclomatic complexity, Halstead metrics, and explicit PL properties (typing, memory safety, concurrency models) all come from Programming Language Theory & software metrics, not surface text (token_count, char_count only).
- **"Most critical technical phase"**
  → Emphasize in your writeup that this step turns raw code into a PL-aware, analyzable dataset that directly supports your hypothesis from Phase 1.