CS421 Final Project Report: Todo-MD

Spring 2025

Xiaoyang Cai (xcai17)

May 16, 2025

Repository: github.com/XiaoyangCai360/cs421_final_project

Overview

In modern software projects, in-code "TODO" and "FIXME" comments proliferate as informal work-items and technical-debt markers. Manually locating and tracking them across multiple files and languages is tedious and error-prone. **Todo-MD** addresses this by statically scanning a codebase—regardless of whether it's Python, JavaScript, Java, C, or Haskell—and producing a single, Markdown-formatted report of every `TODO` and `FIXME` marker, complete with file path, line number, and (in the enhanced version) enclosing function or module context. Version 1 delivers a fast, regex-based extractor; Version 2 layers on a PLY-generated lexer to handle multi-line comments and an AST/regex mapper to attribute markers to their surrounding definitions.

Implementation

Major Tasks and Capabilities

The core capabilities of Todo-MD are:

- **Directory Scanning:** Recursively traverse the project directory (via `os.walk`) to locate source files with extensions `.py`, `.js`, `.java`, `.c`, `.h`, and `.hs`.
- **Marker Extraction (v1):** Perform a line-by-line regular-expression scan (using Python's `re` module) to identify `TODO` and `FIXME` in comment lines and capture file path, line number, and comment text.
- **Report Generation:** Emit a Markdown table listing each marker via `write_markdown()`, producing `report_v1.md`.
- **Advanced Lexing (v2):** Integrate the PLY lexer to tokenize single- and multi-line comment blocks (`#...`, `//...`, `/*...*/`, `--...`) for robust extraction across languages.
- **Context Mapping:** Map each marker to its enclosing definition by traversing Python's AST for `def`/`class` nodes, and by simple regex rules for Haskell signatures/definitions.
- **Enhanced Reporting:** Extend the Markdown output with a "Context" column via `write_markdown_v2()`, yielding `report_v2.md`.

Code Components

The implementation is split into these modules:

`todo_md_v1.py` Implements Version 1: directory traversal, regex scanning, and Markdown report writer.

`todo_lexer.py` Defines PLY token rules for comment types across languages.

`todo_context.py` Provides context mappers: Python AST walker and Haskell regex-based mapper.

`todo_md_v2.py` Orchestrates Version 2: runs the lexer, extracts markers, maps context, and formats the enhanced report.

`Makefile` Automates targets `run-v1`, `run-v2`, `test`, and `clear`.

`tests/` Contains unit, feature, and integration tests written with pytest and CLI checks.

Version 1 Regex Explanation

The key regular expression used by the Version 1 scanner is:

```
(?P<marker>TODO|FIXME)[\s:,-]*(?P<text>.*)
```

This can be read as:

`(?P<marker>TODO|FIXME)` match either the literal `TODO` or `FIXME`, capturing it in the named group `marker`.

`[\s:,-]*` skip over any combination of whitespace (`\s`), colons (`:`), commas (`,`), or hyphens (`-`), zero or more times.

`(?P<text>.*)` capture the rest of the line (any characters, zero or more) in the named group `text`.

Version 2 Regex Explanation

The Version 2 scanner relies on two core regular expressions.

Comment Token Pattern

```
/*(.|\n)*?\*/|//[^\n]*|#[^\n]*|--[^\n]*
```

This single pattern matches:

`/*...*/` C-style block comments

`//...` C++/Java/JS single-line comments

`...` Python single-line comments

`--...` Haskell single-line comments

TODO/FIXME Extraction Pattern

```
(TODO|FIXME)[:\s\-]*(.*)
```

This can be read as:

`(TODO|FIXME)` match the whole word `TODO` or `FIXME`

`[:]*` skip any combination of colons, whitespace, or hyphens

`(.*)` capture the remainder of the comment line

Python Context Mapping

```
def map_python_context(source_code: str) -> dict:
    tree = ast.parse(source_code)
    intervals = []
    class DefVisitor(ast.NodeVisitor):
        def visit_FunctionDef(self, node):
            start = node.lineno
            end   = get_end_lineno(node)
            intervals.append((start, end, node.name))
            self.generic_visit(node)
        def visit_AsyncFunctionDef(self, node):
            start = node.lineno
            end   = get_end_lineno(node)
            intervals.append((start, end, node.name))
            self.generic_visit(node)
        def visit_ClassDef(self, node):
            start = node.lineno
            end   = get_end_lineno(node)
            intervals.append((start, end, node.name))
            self.generic_visit(node)
    DefVisitor().visit(tree)

    lookup = {}
    for start, end, name in intervals:
        for ln in range(start, end+1):
            lookup[ln] = name
    return lookup
```

This does the following:

AST Parsing Builds a Python AST from the source.

Node Visiting Records start and end lines for each `def`, `async def`, and `class`.

Line Mapping Fills a lookup table mapping every line in those intervals to the corresponding name.

Default Lines outside any definition remain unmapped (fall back to `<module>`).

Haskell Context Mapping

```
HS_SIG_RE = re.compile(r"^([A-Za-z_][\w']*)\s*::")
HS_EQ_RE  = re.compile(r"^([A-Za-z_][\w']*)\s+.*=")

def map_haskell_context(source_code: str) -> dict:
    lookup = {}
    current = "<module>"
    for lineno, line in enumerate(source_code.splitlines(), start=1):
        sig = HS_SIG_RE.match(line)
        eq  = HS_EQ_RE.match(line)
        if sig:
            current = sig.group(1)
        elif eq:
            current = eq.group(1)
        lookup[lineno] = current
    return lookup
```

This works by:

Signature Match On lines matching `name :: ...`, set `current = name`.

Definition Match On lines matching `name ... = ...`, set `current = name`.

Line Mapping Assign every line the most recent `current` name, defaulting to `<module>` before the first match.

Status and Comparison to Proposed Goals

- **Fully Implemented (as Proposed):**
  - Version 1 regex-based extraction and Markdown output.
  - Unit tests for core scanners and fixtures.
  - Integration tests and Makefile automation.
- **Enhanced Features Achieved:**
  - PLY-based comment lexing with multi-line support.
  - Python AST context mapping and basic Haskell context mapping.
  - Expanded test suite covering advanced and edge-case scenarios.
- **Deferred / Not Implemented:**
  - Context mapping for C, Java, and JavaScript functions (identified but left for future work).

Example Input and Output

Sample Input Files

Listing 1: foo.py

```
# foo.py
import os        # used
import sys       # unused

# TODO: add error handling for missing directory
def list_dir(path):
    files = os.listdir(path)
    # FIXME: what if path is not a directory?
    return files

def hello():
    # TODO   support greeting in other languages
    print("Hello, world!")
```

Listing 2: bar.js

```
// bar.js

// TODO: migrate this to TypeScript
function greet(name) {
    console.log("Hello, " + name + "!");
}

// Some code without any markers
const x = 5;
```

Listing 3: hsExample.hs

```
-- HsExample.hs
-- TODO: write module header documentation

module HsExample where

-- TODO: implement fibonacci (na ve recursion)
fibonacci :: Int -> Int
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n - 1) + fibonacci (n - 2)

-- FIXME: add memoization for performance
```

Version 1 Output (report_v1.md)

```
# TODO-MD Report

**Scanned directory:** `sample_project`
**Found:** 7 items

| # | File | Line | Marker | Comment |
|---|------|-----:|--------|---------|
| 1 | `sample_project/hsExample.hs` | 2 | TODO | write module header documentation |
```

```
| 2 | ‘sample_project/hsExample.hs‘ | 6 | TODO | implement fibonacci (naïve recursion) |
| 3 | ‘sample_project/hsExample.hs‘ | 12 | FIXME | add memoization for performance |
| 4 | ‘sample_project/foo.py‘ | 6 | TODO | add error handling for missing directory |
| 5 | ‘sample_project/foo.py‘ | 9 | FIXME | what if path is not a directory? |
| 6 | ‘sample_project/foo.py‘ | 13 | TODO | support greeting in other languages |
| 7 | ‘sample_project/bar.js‘ | 3 | TODO | migrate this to TypeScript |
```

Version 2 Output (`report_v2.md`)

```
# TODO-MD v2 Report

**Scanned directory:** ‘sample_project‘
**Found:** 7 items

| # | File | Line | Context | Marker | Comment |
|---|------|-----:|---------|--------|---------|
| 1 | ‘sample_project/hsExample.hs‘ | 2 | <module> | TODO | write module header documentation
| 2 | ‘sample_project/hsExample.hs‘ | 6 | <module> | TODO | implement fibonacci (naïve recursi
| 3 | ‘sample_project/hsExample.hs‘ | 12 | fibonacci | FIXME | add memoization for performance
| 4 | ‘sample_project/foo.py‘ | 6 | <module> | TODO | add error handling for missing directory
| 5 | ‘sample_project/foo.py‘ | 9 | list_dir | FIXME | what if path is not a directory? |
| 6 | ‘sample_project/foo.py‘ | 13 | hello | TODO | support greeting in other languages |
| 7 | ‘sample_project/bar.js‘ | 3 | <module> | TODO | migrate this to TypeScript |
```

Tests

We implemented three layers of tests:

Unit Tests

Individual function tests for:

- `collect_todos()` (v1) on synthetic strings and files.
- `collect_todos_v2()` (v2) on in-memory C-block fixtures.
- `map_line_to_context()` for Python AST nodes and Haskell definitions.

Feature Tests

Fixtures in `tests/fixtures/` cover real-world patterns:

- Single-line markers (Python, JS, Java, Haskell).
- Multi-line block comments.
- Correct context names (e.g. “foo“, “bar“, “safeDiv“, “gcd’“).

Integration Tests

CLI invocation via `subprocess.run` ensures:

- Recursive directory scanning (nested directories, mixed extensions).
- Correct exit codes, report generation, and marker counts.

Running `pytest -q` yields 14 passing tests across unit, feature, and integration suites.

Listing

Key excerpts from the implementation (full code in the repository).

Version 1: Regex-based Scanner and Markdown Writer

Below are the central functions from `todo_md_v1.py` that implement the core regex-driven scanner and Markdown report generator.

Listing 4: v1: Regex-based collector

```
todo_pattern = re.compile(
    r'(?P<marker>TODO|FIXME)[\s:,-]*(?P<text>.*)',
    re.IGNORECASE,
)

# Scan a single file for TODO/FIXME comments
def collect_todos(file_path):
    """
    Returns a list of (lineno, marker, text).
    Only uppercase TODO/FIXME markers are considered.
    """
    todos = []
    try:
        with open(file_path, encoding="utf-8") as f:
            for lineno, line in enumerate(f, start=1):
                m = todo_pattern.search(line)
                if not m:
                    continue
                orig = m.group("marker")
                if not orig or not orig[0].isupper():
                    continue
                text = m.group("text").strip()
                todos.append((lineno, orig.upper(), text))
    except (UnicodeDecodeError, FileNotFoundError):
        pass
    return todos

# Write the collected items into a Markdown file
def write_markdown(items, output_path, scanned_dir):
    with open(output_path, "w", encoding="utf-8") as out:
        out.write("# TODO-MD Report\n\n")
        out.write(f"**Scanned directory:** `{scanned_dir}`  \n")
        out.write(f"**Found:** {len(items)} items\n\n")
```

```
        out.write("| # | File | Line | Marker | Comment |\n")
        out.write("|---|------|-----:|--------|---------|\n")
        for i, (path, lineno, marker, text) in enumerate(items, 1):
            out.write(f"| {i} | `{path}` | {lineno} | {marker} | {text} |\
                n")
    print(f"Wrote Markdown report to {output_path}")
```

Version 2: PLY Lexer and Context Mapping

These snippets from `todo_md_v2.py` drive the enhanced version, using a PLY lexer for comment tokens and mapping each marker to its enclosing context.

Listing 5: v2: PLY lexer

```
# Match C-style block comments (/*...*/), Python    #       , JS    //
         , and Haskell    --       comments
t_COMMENT = r'/\*(.|\n)*?\*/|//[^\n]*|\#[^\n]*|--[^\n]*'

# Track newlines inside comments and elsewhere
def t_newline(t):
    r'\n+'
    t.lexer.lineno += t.value.count('\n')

# Ignore spaces, tabs, and carriage returns
t_ignore = ' \t\r'
```

Listing 6: v2: Python Context Mapping

```
# Python context
def map_python_context(source_code: str) -> dict:
    tree = ast.parse(source_code)
    intervals = []
    class DefVisitor(ast.NodeVisitor):
        def visit_FunctionDef(self, node):
            start = node.lineno
            end   = get_end_lineno(node)
            intervals.append((start, end, node.name))
            self.generic_visit(node)
        def visit_AsyncFunctionDef(self, node):
            start = node.lineno
            end   = get_end_lineno(node)
            intervals.append((start, end, node.name))
            self.generic_visit(node)
        def visit_ClassDef(self, node):
            start = node.lineno
            end   = get_end_lineno(node)
            intervals.append((start, end, node.name))
            self.generic_visit(node)
    DefVisitor().visit(tree)

    lookup = {}
    for start, end, name in intervals:
        for ln in range(start, end+1):
            lookup[ln] = name
```

```
    return lookup
```

Listing 7: v2: Haskell Context Mapping

```
# Haskell context
HS_SIG_RE = re.compile(r"^([A-Za-z_][\w']*)\s*::")
HS_EQ_RE  = re.compile(r"^([A-Za-z_][\w']*)\s+.*=")

def map_haskell_context(source_code: str) -> dict:
    """
    Walk each line, updating `current` whenever we see
    NAME :: ...   or   NAME args = ...
    """
    lookup = {}
    current = "<module>"
    for lineno, line in enumerate(source_code.splitlines(), start=1):
        sig = HS_SIG_RE.match(line)
        eq  = HS_EQ_RE.match(line)
        if sig:
            current = sig.group(1)
        elif eq:
            current = eq.group(1)
        lookup[lineno] = current
    return lookup
```

Listing 8: v2: PLY Lexer + Context Mapping

```
def collect_todos_v2(file_path):
    """
    Lex the entire file to get COMMENT tokens, then regex-find TODO
       markers.
    Returns list of (lineno, marker, comment_text).
    Strips trailing '*/' from block comments.
    """
    try:
        text = open(file_path, encoding="utf-8").read()
    except (UnicodeDecodeError, FileNotFoundError):
        return []

    lexer = build_lexer()
    lexer.input(text)

    comments = []
    for tok in lexer:
        if tok.type == "COMMENT":
            comments.append((tok.lineno, tok.value))

    results = []
    for lineno, comment in comments:
        for m in TODO_PATTERN.finditer(comment):
            marker = m.group(1).upper()
            comment_text = m.group(2).strip()
            # remove trailing block comment delimiters if present
            comment_text = TRAILER_PATTERN.sub("", comment_text)
```

```
            results.append((lineno, marker, comment_text))
    return results
```

Discussions and Future Works

While Todo-MD now handles single- and multi-line comments across several languages and maps markers to Python and Haskell contexts, there are clear avenues for enhancement. Future work could include:

- **Haskell Mapping Limitation:** The current Haskell mapper binds comments only if they appear on or after a function's signature; comments placed immediately above the signature (the common style) remain mapped to 'module', so we plan to enhance the mapper to look ahead and associate pre-definition comments with the function that follows.
- **Language Coverage:** Adding context mappers for C, Java, and JavaScript functions and methods using lightweight regex or parser generators.
- **Configuration File:** Allow users to specify custom comment markers (e.g., `NOTE`, `XXX`) and file extensions via a TOML or YAML config.
- **Output Formats:** Support JSON or HTML output in addition to Markdown, enabling integration with dashboards or web interfaces.

References

[1] M. Beckman, *CS 421: Programming Languages*, University of Illinois at Urbana–Champaign, Spring 2025, `https://cs421-sp25-web.pages.dev/`.

[2] D. Beazley, *PLY (Python Lex-Yacc) Documentation*, `https://www.dabeaz.com/ply/`.

[3] Python Software Foundation, *ast — Abstract Syntax Trees*, Python 3.9.6 documentation, `https://docs.python.org/3/library/ast.html`.

[4] OpenAI, *ChatGPT*, May 2025, `https://chat.openai.com/`.