LAPORAN TUGAS PROGRAMING LANGUAGE PRAGMATICS

2B-D4

Programming language pragmatic



Disusun oleh:

|  |  |
| --- | --- |
| Fariz Dhifa Fakhriza | 241524040 |
| Aulia Rachmad Adiwinarrta | 241524034 |
| Idotoho Reimon Simanjuntak | 241524047 |

**JURUSAN TEKNIK KOMPUTER DAN INFORMATIKA PROGRAM STUDI SARJANA TERAPAN TEKNIK INFORMATIKA POLITEKNIK NEGERI BANDUNG**

**2025**

## BAB 1: PENDAHULUAN

### 1.1 Latar Belakang

Dalam industri pengembangan perangkat lunak, kualitas kode adalah salah satu faktor utama yang menentukan keberhasilan jangka panjang sebuah proyek. Kode yang "bersih" (Clean Code)—kode yang mudah dibaca, dipahami, dan dipelihara—secara signifikan mengurangi biaya *maintenance*, menekan jumlah *bug*, dan mempermudah kolaborasi tim. Namun, peninjauan kode (code review) secara manual untuk memastikan kepatuhan terhadap standar *clean code* adalah proses yang subjektif, memakan waktu, dan rentan terhadap kesalahan manusia (human error).

Salah satu pendekatan untuk mengatasi masalah ini adalah melalui *Static Code Analysis* atau *Linter*. Alat-alat ini menganalisis kode sumber tanpa menjalankannya untuk mendeteksi potensi masalah, *bug*, atau pelanggaran gaya penulisan.

ANTLR (ANother Tool for Language Recognition) adalah *parser generator* yang kuat untuk membaca, memproses, dan menerjemahkan teks terstruktur atau bahasa komputer. Dengan ANTLR, kita dapat membangun alat kustom untuk memahami struktur bahasa pemrograman, seperti bahasa C, dan menerapkan aturan spesifik di atasnya.

Dalam proyek ini, kami mengembangkan sebuah *Domain-Specific Language* (DSL) yang berfungsi sebagai *linter* sederhana. Domain yang kami fokuskan adalah "Manajemen Kualitas Kode" dengan menerapkan beberapa aturan *clean code* dasar pada *file* kode sumber bahasa C.

### 1.2 Rumusan Masalah

Berdasarkan latar belakang di atas, rumusan masalah yang akan diselesaikan adalah:

1. Bagaimana merancang sebuah sistem yang dapat mem-parsing dan memahami struktur kode bahasa C menggunakan ANTLR?
2. Bagaimana mengimplementasikan logika untuk mendeteksi pelanggaran aturan *clean code* (code smells) secara otomatis pada bahasa C?
3. Bagaimana aturan *clean code* yang spesifik (seperti jumlah parameter, panjang fungsi, dan fungsi kosong) dapat diterapkan menggunakan ANTLR Visitor Pattern?

### 1.3 Tujuan Proyek

Tujuan dari proyek ini adalah:

1. Membangun sebuah aplikasi *linter* berbasis *console* menggunakan ANTLR 4 dan Python.
2. Menggunakan grammar ANTLR yang sudah ada untuk bahasa C (C.g4) sebagai dasar parser.
3. Mengimplementasikan *Visitor* kustom untuk menelusuri *Parse Tree* dan mengekstraksi data.
4. Menerapkan logika analisis untuk tiga aturan *clean code* utama:
   * Fungsi tidak boleh memiliki lebih dari 5 parameter.
   * Fungsi tidak boleh memiliki lebih dari 20 baris kode (statement).
   * Fungsi tidak boleh kosong (tidak memiliki *statement*).
5. Menyediakan laporan yang jelas dan interaktif bagi pengguna, baik untuk satu *file* maupun satu proyek (folder).

### 1.4 Batasan Masalah

Proyek ini memiliki batasan sebagai berikut:

1. Bahasa yang dianalisis hanya bahasa C (sesuai standar grammar C.g4).
2. Analisis tidak menangani *preprocessor* C (seperti #include atau #define).
3. Aturan yang dicek terbatas pada tiga aturan yang disebutkan di tujuan.
4. Aplikasi hanya berjalan di *command-line* (CLI) dan tidak memiliki GUI.
5. Program hanya mendeteksi dan melaporkan pelanggaran, tidak melakukan perbaikan kode otomatis.

<div style="page-break-after: always;"></div>

## BAB 2: LANDASAN TEORI

### 2.1 ANTLR 4

ANTLR (ANother Tool for Language Recognition) adalah sebuah *parser generator* yang ditulis dalam Java. ANTLR digunakan untuk membangun *compiler*, *interpreter*, dan alat bantu bahasa lainnya. ANTLR mengambil *file* grammar (.g4) yang mendefinisikan sintaks sebuah bahasa, dan secara otomatis menghasilkan serangkaian *file* kode (seperti Python, Java, C#) yang membentuk komponen-komponen berikut:

* **Lexer:** Bertugas memecah teks input menjadi unit-unit kecil yang disebut *token* (kosakata).
* **Parser:** Menerima *token* dari Lexer dan memverifikasi apakah urutannya sesuai dengan aturan sintaks dalam grammar. Jika valid, Parser akan membangun struktur data pohon yang disebut *Parse Tree* (juga dikenal sebagai *Abstract Syntax Tree* atau AST).
* **Parse Tree:** Representasi hierarkis dari kode sumber yang mencerminkan struktur logisnya.

### 2.2 Visitor Pattern

ANTLR menyediakan dua mekanisme utama untuk menelusuri *Parse Tree*: *Listeners* dan *Visitors*. Dalam proyek ini, kami menggunakan **Visitor Pattern**.

Berbeda dengan *Listener* yang bekerja secara pasif (dipanggil otomatis oleh *walker*), *Visitor* bekerja secara aktif (eksplisit). Kita harus secara manual memanggil method visit() untuk berpindah ke node anak. Keunggulannya adalah kita memiliki kontrol penuh atas proses penelusuran. Ini sangat ideal untuk kasus di mana kita perlu mengambil data dari node anak dan mengembalikannya ke node induk (misalnya, menghitung total baris).

### 2.3 Static Code Analysis (Linter)

Analisis Kode Statis adalah proses menganalisis kode sumber perangkat lunak tanpa mengeksekusinya. Tujuannya adalah untuk menemukan potensi *bug*, kerentanan keamanan, atau pelanggaran terhadap kaidah penulisan kode (*coding standards*). Alat yang melakukan ini disebut *Linter*.

*Linter* yang kami bangun berfokus pada deteksi "Code Smells"—indikator adanya masalah desain yang lebih dalam pada kode.

### 2.4 Aturan Clean Code (Domain)

Proyek ini berfokus pada domain "Manajemen Kualitas Kode" dengan mengacu pada prinsip *Clean Code* yang dipopulerkan oleh Robert C. Martin. Aturan yang kami implementasikan adalah:

1. **Fungsi Terlalu Panjang (Long Function):** Fungsi yang terlalu panjang (di proyek ini > 20 baris) biasanya menandakan bahwa fungsi tersebut melakukan terlalu banyak hal (melanggar *Single Responsibility Principle*).
2. **Daftar Parameter Panjang (Long Parameter List):** Fungsi dengan banyak parameter (di proyek ini > 5) sulit untuk digunakan, diuji, dan dipahami.
3. **Fungsi Kosong (Empty Function):** Fungsi yang didefinisikan tetapi tidak memiliki isi (0 baris) adalah kode mati (*dead code*) yang mengotori *codebase*.

<div style="page-break-after: always;"></div>

## BAB 3: PERANCANGAN DAN IMPLEMENTASI

### 3.1 Arsitektur Sistem

Arsitektur alat kami mengikuti alur kerja standar ANTLR yang digabungkan dengan antarmuka pengguna CLI:

1. **Input (CLI):** Pengguna memilih mode (Scan File / Scan Folder).
2. **File Reader:** Sistem membaca *file* .c dari *path* yang diberikan.
3. **Lexer (CLexer.py):** Teks *file* diubah menjadi *stream* token.
4. **Parser (CParser.py):** *Token* disusun menjadi *Parse Tree*.
5. **Visitor (DetailedCAnalyzer):** *Visitor* kustom kami menelusuri pohon, menerapkan aturan *clean code* pada setiap node functionDefinition.
6. **Output (CLI):** Hasil analisis dan laporan pelanggaran dicetak ke *console*.

### 3.2 Perancangan Grammar (C.g4)

Kami tidak menulis grammar bahasa C dari nol, karena bahasa C sangat kompleks dan telah terstandarisasi. Kami menggunakan grammar C.g4 resmi yang disediakan oleh repositori antlr/grammars-v4 di GitHub.

Fokus kami adalah mengidentifikasi *rule* (aturan) kunci dalam grammar tersebut yang relevan dengan domain kami:

1. **functionDefinition:** Ini adalah *rule* utama yang mewakili satu fungsi C secara keseluruhan. Ini menjadi titik masuk *Visitor* kami untuk memulai analisis per fungsi.
2. **compoundStatement:** Mewakili *body* (isi) dari sebuah fungsi, yang ditandai dengan { ... }.
3. **blockItemList dan blockItem:** Mewakili daftar *statement* atau baris kode di dalam compoundStatement. Dengan menghitung jumlah blockItem, kami mendapatkan total "Line of Code" (LoC).
4. **parameterTypeList dan parameterDeclaration:** Mewakili daftar parameter di dalam deklarasi fungsi. Dengan menghitung jumlah parameterDeclaration, kami mendapatkan jumlah parameter.

### 3.3 Implementasi Visitor (DetailedCAnalyzer)

Inti dari logika proyek ini terletak pada *class* DetailedCAnalyzer yang mewarisi CVisitor.

#### 3.3.1 Menargetkan Fungsi

Kami meng-override method visitFunctionDefinition. Method ini dipanggil setiap kali parser menemukan sebuah fungsi.

class DetailedCAnalyzer(CVisitor):  
 def \_\_init\_\_(self):  
 self.violations = []  
 self.function\_count = 0  
 self.clean\_functions = 0  
   
 # ... (helper pencari nama & parameter) ...  
  
 def visitFunctionDefinition(self, ctx):  
 self.function\_count += 1  
 start\_line = ctx.start.line  
   
 # A. Ambil Data  
 func\_name = self.get\_function\_name(ctx)  
 param\_count = self.count\_parameters(ctx)  
   
 # B. Hitung Baris Kode (Statement)  
 body = ctx.compoundStatement()  
 line\_count = 0  
 if body.blockItemList():  
 line\_count = len(body.blockItemList().blockItem())  
  
 # C. Logika Pengecekan Rules  
 is\_clean = True  
   
 if line\_count > 20:  
 self.add\_violation(start\_line, ..., func\_name, "Fungsi terlalu panjang", line\_count, 20)  
 is\_clean = False  
  
 if param\_count > 5:  
 self.add\_violation(start\_line, ..., func\_name, "Terlalu banyak parameter", param\_count, 5)  
 is\_clean = False  
   
 if line\_count == 0:  
 self.add\_violation(start\_line, ..., func\_name, "Fungsi kosong", 0, "> 0")  
 is\_clean = False  
  
 if is\_clean:  
 self.clean\_functions += 1  
   
 return self.visitChildren(ctx)

#### 3.3.2 Navigasi Grammar C yang Kompleks

Tantangan terbesar adalah mengekstrak nama fungsi dan parameter, karena struktur *declarator* C sangat rumit (misalnya, int (\*nama\_fungsi)(int a)). Kami membuat fungsi *helper* rekursif (get\_function\_name dan count\_parameters) untuk menelusuri struktur directDeclarator dan parameterTypeList hingga menemukan Identifier dan daftar parameterDeclaration.

### 3.4 Implementasi Antarmuka (main.py)

Antarmuka pengguna (UI) dibuat interaktif menggunakan loop while True sederhana yang berfungsi sebagai *main menu*.

def main():  
 while True:  
 clear\_screen() # Membersihkan layar  
 print\_banner()  
 print("1. Scan File Spesifik")  
 print("2. Scan Semua File di Folder (Batch Mode)")  
 print("3. Keluar")  
   
 pilihan = input("\nPilihan Anda (1-3): ")  
   
 if pilihan == '1':  
 menu\_scan\_file()  
 elif pilihan == '2':  
 menu\_scan\_folder()  
 elif pilihan == '3':  
 print("Sampai jumpa!")  
 break  
 else:  
 print("Pilihan tidak valid.")  
 input("Tekan Enter...")

Fungsi menu\_scan\_file meminta input nama *file* spesifik, sedangkan menu\_scan\_folder menggunakan modul glob untuk mencari semua *file* \*.c di direktori yang dituju, memberikan gambaran "Manajemen Proyek" yang lebih luas.

<div style="page-break-after: always;"></div>

## BAB 4: PENGUJIAN DAN ANALISIS

### 4.1 Skenario Pengujian

Untuk menguji fungsionalitas *linter*, kami membuat sebuah *file* test.c yang sengaja diisi dengan pelanggaran terhadap ketiga aturan yang telah ditetapkan.

**File: test.c**

#include <stdio.h>  
  
// 1. Fungsi Bersih (Clean)  
int tambah(int a, int b) {  
 return a + b;  
}  
  
// 2. Pelanggaran: Parameter Kebanyakan (7 parameter)  
void fungsiRibet(int a, int b, int c, int d, int e, int f, int g) {  
 printf("Ini kebanyakan parameter");  
}  
  
// 3. Pelanggaran: Baris Kebanyakan (21 baris)  
void fungsiPanjang() {  
 int a = 1; int b = 2; int c = 3; int d = 4; int e = 5;  
 a = b + c; b = c + d; c = d + e; d = e + a; e = a + b;  
 a = b + c; b = c + d; c = d + e; d = e + a; e = a + b;  
 a = b + c; b = c + d; c = d + e; d = e + a; e = a + b;  
 printf("Selesai");  
}  
  
// 4. Pelanggaran: Fungsi Kosong  
void fungsiHantu() {  
}

### 4.2 Hasil Pengujian

Kami menjalankan mode "Scan File Spesifik" dan "Scan Folder" pada *file* uji tersebut.

#### 4.2.1 Mode Scan File Spesifik

Saat memilih Opsi 1 dan memasukkan test.c, program mengeluarkan laporan detail:

>>> Scanning: test.c  
 [FAIL] Line 8: fungsiRibet -> Terlalu banyak parameter (Actual: 7 | Limit: 5)  
 [FAIL] Line 13: fungsiPanjang -> Fungsi terlalu panjang (Actual: 21 | Limit: 20)  
 [FAIL] Line 20: fungsiHantu -> Fungsi kosong (Actual: 0 | Limit: > 0)  
 --> File Health Score: 25%

*(Catatan: Fungsi tambah lolos tanpa laporan, sesuai harapan).*

#### 4.2.2 Mode Scan Folder

Saat memilih Opsi 2 (dan test.c adalah satu-satunya *file* di folder), program mengeluarkan ringkasan proyek:

============================================================  
 PROJECT SUMMARY REPORT  
============================================================  
Total Files Analyzed : 1  
Total Functions Found : 4  
Total Violations : 3  
Project Health Score : 25.00%

### 4.3 Analisis Hasil

Hasil pengujian menunjukkan bahwa sistem berhasil 100% dalam mendeteksi ketiga jenis pelanggaran *clean code* yang telah didefinisikan.

1. **Deteksi Parameter:** *Visitor* berhasil menavigasi struktur parameterTypeList dan menghitung 7 parameter pada fungsiRibet.
2. **Deteksi Panjang Fungsi:** *Visitor* berhasil menghitung 21 blockItem di dalam fungsiPanjang.
3. **Deteksi Fungsi Kosong:** *Visitor* berhasil mendeteksi blockItemList yang kosong pada fungsiHantu dan melaporkan line\_count = 0.
4. **Akurasi Laporan:** Laporan menunjukkan nama fungsi dan nomor baris yang tepat, mempermudah *developer* untuk menemukan dan memperbaiki masalah.
5. **Performa:** Untuk *file* pengujian, analisis berjalan instan (< 0.1 detik).

<div style="page-break-after: always;"></div>

## BAB 5: PENUTUP

### 5.1 Kesimpulan

Berdasarkan perancangan, implementasi, dan pengujian yang telah dilakukan, dapat ditarik kesimpulan sebagai berikut:

1. ANTLR 4, dikombinasikan dengan Python, terbukti sebagai *framework* yang sangat efektif untuk membangun alat analisis kode statis (*linter*) kustom.
2. Penggunaan grammar bahasa C (C.g4) yang sudah ada memungkinkan proyek untuk fokus pada logika analisis (domain) alih-alih menghabiskan waktu pada perancangan sintaks bahasa yang kompleks.
3. *Visitor Pattern* memberikan kontrol penuh yang diperlukan untuk menavigasi *Parse Tree* yang kompleks dari bahasa C dan berhasil diimplementasikan untuk mengekstrak data nama fungsi, jumlah parameter, dan jumlah baris kode.
4. Aplikasi *linter* yang dikembangkan telah berhasil memenuhi semua tujuan proyek, dengan mampu mendeteksi secara akurat pelanggaran aturan *clean code* yang ditetapkan (fungsi panjang, parameter panjang, fungsi kosong) dan menyajikannya dalam antarmuka CLI yang interaktif.

### 5.2 Saran

Proyek ini dapat dikembangkan lebih lanjut ke beberapa arah, di antaranya:

1. **Penambahan Aturan:** Menambah aturan *clean code* lain, seperti *Naming Conventions* (cth: variabel harus *snake\_case*), kompleksitas siklomatik, atau deteksi *magic numbers*.
2. **Dukungan Bahasa Lain:** Mengadaptasi *Visitor* untuk bekerja dengan grammar bahasa lain yang tersedia di repositori ANTLR, seperti Java atau Python.
3. **Integrasi:** Mengubah program menjadi *tool* yang dapat diintegrasikan dengan *pipeline* CI/CD atau *code editor* (seperti VS Code Extension) untuk memberikan *feedback* *real-time*.
4. **Perbaikan Otomatis (Auto-fix):** Menambahkan fungsionalitas untuk tidak hanya mendeteksi, tetapi juga menawarkan perbaikan otomatis untuk beberapa pelanggaran sederhana.

## DAFTAR PUSTAKA

* Parr, Terence. (2013). *The Definitive ANTLR 4 Reference*. Pragma ADE.
* Martin, Robert C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.
* Repositori ANTLR Grammars-v4. (2023). *Grammar Bahasa C*. Diakses dari: https://github.com/antlr/grammars-v4/tree/master/c

<div style="page-break-after: always;"></div>

## LAMPIRAN

### Kode Sumber Utama (main.py)

import sys  
import os  
import glob  
from antlr4 import \*  
from CLexer import CLexer  
from CParser import CParser  
from CVisitor import CVisitor  
  
# ==========================================  
# 1. CLASS PEWARNAAN & UTILITY TAMPILAN  
# ==========================================  
class Colors:  
 HEADER = '\033[95m'  
 OKBLUE = '\033[94m'  
 OKCYAN = '\033[96m'  
 OKGREEN = '\033[92m'  
 WARNING = '\033[93m'  
 FAIL = '\033[91m'  
 ENDC = '\033[0m'  
 BOLD = '\033[1m'  
  
def clear\_screen():  
 """Membersihkan layar terminal (Support Windows & Linux/Mac)"""  
 os.system('cls' if os.name == 'nt' else 'clear')  
  
def print\_banner():  
 print(f"{Colors.OKCYAN}{'='\*60}")  
 print(f" CLEAN CODE ANALYZER FOR C (DSL ENGINE)")  
 print(f"{'='\*60}{Colors.ENDC}")  
  
# ==========================================  
# 2. VISITOR / ANALYZER LOGIC  
# ==========================================  
class DetailedCAnalyzer(CVisitor):  
 def \_\_init\_\_(self):  
 self.violations = [] # Menyimpan daftar pelanggaran  
 self.function\_count = 0 # Menghitung total fungsi  
 self.clean\_functions = 0 # Menghitung fungsi yang bersih  
  
 def add\_violation(self, line, col, func\_name, message, actual\_val, limit\_val):  
 """Helper untuk mencatat error ke dalam list"""  
 report = {  
 "line": line,  
 "col": col,  
 "function": func\_name,  
 "message": message,  
 "detail": f"Actual: {actual\_val} | Limit: {limit\_val}"  
 }  
 self.violations.append(report)  
  
 def get\_function\_name(self, ctx):  
 """Mencari nama fungsi di dalam struktur Declarator C yang rumit"""  
 decl = ctx.declarator()  
 curr = decl.directDeclarator()  
   
 while curr:  
 if curr.Identifier():  
 return curr.Identifier().getText()  
 if curr.declarator():  
 curr = curr.declarator().directDeclarator()  
 elif curr.directDeclarator():  
 curr = curr.directDeclarator()  
 else:  
 break  
 return "UnknownFunction"  
  
 def count\_parameters(self, ctx):  
 """Menghitung jumlah parameter fungsi"""  
 decl = ctx.declarator()  
 curr = decl.directDeclarator()  
   
 while curr:  
 if curr.parameterTypeList():  
 param\_list\_ctx = curr.parameterTypeList().parameterList()  
 return len(param\_list\_ctx.parameterDeclaration())  
 if curr.directDeclarator():  
 curr = curr.directDeclarator()  
 else:  
 break  
 return 0  
  
 def visitFunctionDefinition(self, ctx):  
 """Method utama yang dipanggil saat Parser menemukan fungsi"""  
 self.function\_count += 1  
 start\_line = ctx.start.line  
 start\_col = ctx.start.column  
   
 func\_name = self.get\_function\_name(ctx)  
 param\_count = self.count\_parameters(ctx)  
   
 body = ctx.compoundStatement()  
 line\_count = 0  
 if body.blockItemList():  
 line\_count = len(body.blockItemList().blockItem())  
  
 is\_clean = True  
   
 # Rule 1: Maksimal 20 Baris  
 if line\_count > 20:  
 self.add\_violation(start\_line, start\_col, func\_name, "Fungsi terlalu panjang", line\_count, 20)  
 is\_clean = False  
 # Rule 2: Maksimal 5 Parameter  
 if param\_count > 5:  
 self.add\_violation(start\_line, start\_col, func\_name, "Terlalu banyak parameter", param\_count, 5)  
 is\_clean = False  
 # Rule 3: Fungsi Kosong (0 baris)  
 if line\_count == 0:  
 self.add\_violation(start\_line, start\_col, func\_name, "Fungsi kosong (Empty Body)", 0, "> 0")  
 is\_clean = False  
  
 if is\_clean:  
 self.clean\_functions += 1  
   
 return self.visitChildren(ctx)  
  
# ==========================================  
# 3. FUNGSI CORE (PROCESSOR)  
# ==========================================  
def analyze\_single\_file(file\_path):  
 """Membaca file .c, menjalankan parser, dan mencetak hasil per file"""  
 print(f"\n{Colors.HEADER}>>> Scanning: {file\_path}{Colors.ENDC}")  
   
 try:  
 input\_stream = FileStream(file\_path)  
 lexer = CLexer(input\_stream)  
 stream = CommonTokenStream(lexer)  
 parser = CParser(stream)  
   
 tree = parser.compilationUnit()  
   
 analyzer = DetailedCAnalyzer()  
 analyzer.visit(tree)  
   
 if analyzer.violations:  
 for v in analyzer.violations:  
 print(f" {Colors.FAIL}[FAIL]{Colors.ENDC} Line {v['line']}: {v['function']} -> {v['message']} ({v['detail']})")  
 else:  
 print(f" {Colors.OKGREEN}[PASS] File Bersih!{Colors.ENDC}")  
   
 return analyzer  
  
 except Exception as e:  
 print(f" {Colors.FAIL}[ERROR] Gagal membaca/parsing file: {e}{Colors.ENDC}")  
 return None  
  
# ==========================================  
# 4. UI MENUS  
# ==========================================  
  
def menu\_scan\_file():  
 """Menu untuk scan satu file spesifik"""  
 while True:  
 target = input(f"\n{Colors.BOLD}Masukkan nama file (cth: test.c) atau 'b' untuk kembali: {Colors.ENDC}")  
 if target.lower() == 'b':  
 break   
 if not os.path.exists(target):  
 print(f"{Colors.FAIL}File '{target}' tidak ditemukan!{Colors.ENDC}")  
 continue  
   
 analyzer = analyze\_single\_file(target)  
 if analyzer and analyzer.function\_count > 0:  
 score = (analyzer.clean\_functions / analyzer.function\_count) \* 100  
 print(f" --> File Health Score: {int(score)}%")  
  
def menu\_scan\_folder():  
 """Menu untuk scan massal dalam folder"""  
 folder = input(f"\n{Colors.BOLD}Masukkan path folder (Enter untuk folder saat ini): {Colors.ENDC}")  
 if folder.strip() == "":  
 folder = "."  
   
 print(f"Mencari file .c di '{folder}'...")  
 list\_files = glob.glob(os.path.join(folder, "\*.c"))  
   
 if not list\_files:  
 print(f"{Colors.WARNING}Tidak ada file .c di folder tersebut.{Colors.ENDC}")  
 input("Tekan Enter untuk kembali...")  
 return  
  
 total\_files = len(list\_files)  
 total\_funcs\_project = 0  
 clean\_funcs\_project = 0  
 total\_violations = 0  
  
 for f in list\_files:  
 res = analyze\_single\_file(f)  
 if res:  
 total\_funcs\_project += res.function\_count  
 clean\_funcs\_project += res.clean\_functions  
 total\_violations += len(res.violations)  
   
 print(f"\n{Colors.HEADER}{'='\*60}")  
 print(f" PROJECT SUMMARY REPORT")  
 print(f"{'='\*60}{Colors.ENDC}")  
 print(f"Total Files Analyzed : {total\_files}")  
 print(f"Total Functions Found : {total\_funcs\_project}")  
 print(f"Total Violations : {total\_violations}")  
   
 if total\_funcs\_project > 0:  
 score = (clean\_funcs\_project / total\_funcs\_project) \* 100  
 else:  
 score = 0  
   
 color\_score = Colors.OKGREEN if score == 100 else (Colors.WARNING if score > 70 else Colors.FAIL)  
 print(f"Project Health Score : {color\_score}{score:.2f}%{Colors.ENDC}")  
   
 input(f"\n{Colors.BOLD}[Tekan Enter untuk kembali ke Menu Utama]{Colors.ENDC}")  
  
# ==========================================  
# 5. MAIN ENTRY POINT  
# ==========================================  
def main():  
 while True:  
 clear\_screen()  
 print\_banner()  
 print("1. Scan File Spesifik")  
 print("2. Scan Semua File di Folder (Batch Mode)")  
 print("3. Keluar")  
   
 pilihan = input(f"\n{Colors.BOLD}Pilihan Anda (1-3): {Colors.ENDC}")  
   
 if pilihan == '1':  
 menu\_scan\_file()  
 elif pilihan == '2':  
 menu\_scan\_folder()  
 elif pilihan == '3':  
 print("Sampai jumpa!")  
 break  
 else:  
 print("Pilihan tidak valid.")  
 input("Tekan Enter...")  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()