Assignment 1: Deep Dive into C4 - Understanding a Self-Interpreting C Compiler

Due Date: March, 03, 2025, 23:59

Objective:

The goal of this assignment is to thoroughly analyze and understand the structure, functionality, and implementation details of the C4 compiler. By the end of this assignment, you will be able to explain how the code works, identify its key components, and reason about its design decisions.

Tasks:

1. Annotate the Code:

- Download the C4 code and add detailed comments to **every function, block, and non-trivial line of code**.
- Your comments should explain:
- What the code does.
- Why it is written in a particular way.
- How it fits into the overall structure of the compiler.
- Save the annotated code as `c4_annotated.c`.

2. Draw a Detailed Diagram:

- Create a diagram that represents the **architecture and data flow** of the C4 compiler. Include:
- The main components (e.g., lexer, parser, virtual machine, etc.).
- How data moves between these components.
- Key data structures (e.g., tokens, instructions, stack, etc.).
- Save the diagram as `c4_diagram.png` or `c4_diagram.pdf`.

3. Collect Code Statistics:

- Use static and dynamic analysis tools to collect the following statistics about the C4 code:
- Lines of code (LOC) for the entire program and for individual functions.
- Cyclomatic complexity of each function.
- Number of functions and their sizes (in LOC).
- Number of global variables and their usage.
- Number of unique tokens and their frequency.
- Number of branches, loops, and their nesting levels.
- Memory usage patterns (e.g., stack vs. heap allocation).
- Write a summary of your findings in a file named `c4_statistics.txt`.

4. Explain Key Algorithms:

- Write a short report (1-2 pages) explaining the following algorithms or concepts used in C4:
- The **lexical analysis** process: How does the code identify and tokenize input?
- The **parsing** process: How does the code construct an abstract syntax tree (AST) or equivalent representation?
- The **virtual machine** implementation: How does the code execute the compiled instructions?
- The **memory management** approach: How does the code handle memory allocation and deallocation?
- Save the report as `c4_report.pdf`.

5. Answer Conceptual Questions:

- Provide written answers to the following questions:

1. What is the purpose of the `next()` function, and how does it contribute to the compilation process?
2. How does C4 handle **symbol resolution** (e.g., variables, functions)?
3. What are the limitations of C4 as a compiler? What features of C does it not support?
4. How does C4 achieve **self-hosting** (i.e., compiling itself)? What are the implications of this design?

- Save the answers in a file named `c4_answers.txt`.

6. Experiment with the Code (Optional; Bonus 10%):

- Compile and run the C4 compiler on your local machine.
- Modify the code to **add a new feature** or **change an existing behavior**. For example:
- Add support for a new operator (e.g., `%` for modulus).
- Change the way the virtual machine handles a specific instruction.
- Write a short explanation of your changes and how they affect the compiler's behavior.
- Save the modified code as `c4_modified.c` and the explanation as `c4_modification_explanation.txt`.

Deliverables:

1. `c4_annotated.c`: Annotated C4 code with detailed comments.
2. `c4_diagram.png` or `c4_diagram.pdf`: Diagram of the C4 architecture and data flow.
3. `c4_statistics.txt`: Summary of code statistics collected using analysis tools.
4. `c4_report.pdf`: Short report explaining key algorithms and concepts.
5. `c4_answers.txt`: Written answers to the conceptual questions.
6. `c4_modified.c`: Modified version of the C4 code with your changes.
7. `c4_modification_explanation.txt`: Explanation of your changes and their impact.

Submission Instructions:

1. Create a GitHub repository named `c4_analysis` and upload all deliverables to the repository.

2. Compress the repository into a zip file named `c4_analysis.zip`.
3. Submit the zip file on Blackboard.

Grading Criteria:

- **Depth of Understanding**: Comments, diagrams, and explanations should demonstrate a thorough understanding of the code.
- **Originality**: Work must be your own. Generative AI should not be used to complete the assignment.
- **Clarity**: Diagrams, comments, and explanations should be clear and easy to follow.
- **Correctness**: Your modifications to the code should work as intended and be well-explained.
- **Completeness**: All deliverables must be submitted and meet the requirements.

This assignment will challenge you to think critically and engage deeply with the code, setting a strong foundation for the rest of the course. Good luck!

Helpful Tools: Here are some tools you can use to collect the above statistics in #3:

1. **Static Analysis Tools**:

- **Cppcheck**: A static analysis tool for C/C++ that can detect issues like memory leaks, unused variables, and complex code.
- **Clang Static Analyzer**: A powerful tool for analyzing C/C++ code, including metrics like cyclomatic complexity.
- **SourceMonitor**: A tool for measuring code metrics such as LOC, complexity, and function size.
- **Lizard**: A lightweight tool for analyzing code complexity and generating metrics.

2. **Dynamic Analysis Tools**:

- **Valgrind**: A tool for analyzing memory usage, detecting leaks, and profiling performance.
- **Gprof**: A profiling tool for measuring the execution time of functions.
- **strace/ltrace**: Tools for tracing system calls and library calls, respectively, which can help understand runtime behavior.

3. **Visualization Tools**:

- **Doxygen**: Generates documentation and call graphs from annotated code.
- **CodeViz**: A tool for visualizing function call graphs.
- **Gephi**: A graph visualization tool that can be used to visualize call graphs or dependency graphs.

4. **Compiler Flags**:

- Use compiler flags like `-Wall`, `-Wextra`, and `-g` to enable warnings and debugging information, which can help identify potential issues and understand the code better.