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 next() function drives lexical analysis, forming the backbone of the compiler's front end. It reads the source code character by character using a pointer p, with lp marking the start of the current line. The lexer skips whitespace and newline characters; upon encountering a newline (\n), it increments the line counter to track progress. When debugging is enabled via the src flag, it prints the current line from lp to p along with any intermediate code or emitted instructions, including opcode mnemonics, to illustrate how the source is translated into lower-level instructions. After processing, lp updates to p for the next line. The function also bypasses comments and preprocessor directives. For instance, when it finds a # (indicating a preprocessor directive), it ignores the rest of the line. Identifiers starting with alphabetic characters or underscores are recognized by reading until a non-alphanumeric character appears. As it reads an identifier, the lexer computes a hash by multiplying the current hash by 147 and adding the ASCII value of each character. The hash is finalized by left-shifting and incorporating the identifier's length, then used with a `memcmp` check to search the symbol table; if found, the token is reused, otherwise a new entry is created. Numeric literals are processed by converting digit sequences into integers. String literals, enclosed in single or double quotes, are copied into a dedicated data area with proper handling of escape sequences like \n. Operators and punctuation are recognized through conditionals; for example, the lexer distinguishes between = and == by checking the next character. Each token—identifier, literal, or operator—is mapped to an enumerated constant such as Num, Id, or Assign, ensuring a standardized token set for the parser. This systematic approach simplifies later parsing stages and enhances compiler efficiency. Debugging output further aids developers in understanding code translation during compilation.

The **parsing** process: How does the code construct an abstract syntax tree (AST) or equivalent representation?

The parsing process in the code uses a recursive descent strategy to directly translate source code into a linear sequence of intermediate code instructions stored in an array e. Instead of building a traditional abstract syntax tree (AST), the parser combines syntactic analysis with code generation, producing a stream of opcodes that represent the program's logic. The expr() function is central to this approach, handling various expressions by checking the token type. For numeric literals "Num", it emits an "IMM" (Immediate) opcode followed by the value "ival". For string literals, it emits code to load the address of the string from the data segment. When an identifier "Id" is detected, the parser checks if it's a function call by looking for an opening parenthesis. If so, it processes the arguments and emits a "JSR" opcode for user-defined functions or a system call opcode for library functions. Operator precedence is managed through a loop "while (tk >= lev)", which processes binary operators like assignment "Assign", arithmetic "Add, Sub, Mul, Div", and logical operations, emitting corresponding opcodes and ensuring correct type handling, especially for pointer arithmetic.

The “stmt()” function parses statements, including control structures like “if” and “while”. For “if” statements, it parses the condition using “expr(Assign)” and emits a branch instruction “BZ” to direct execution based on the condition. For while loops, it records the current code pointer, processes the condition, and emits branch “BZ” and jump “JMP” instructions to enable looping. Function definitions are parsed by emitting an “ENT” opcode to set up the stack frame and a “LEV” opcode to exit the function. This integration of parsing and code generation creates an efficient pipeline that converts high-level syntax into opcodes, interpreted by a virtual machine in main(). The design is compact and effective, avoiding the need for an AST while handling operator precedence, function calls, type casting, and control flow directly within the parsing routines.

The **virtual machine** implementation: How does the code execute the compiled Instructions?

The virtual machine in the code executes the compiled instructions through a well-structured fetch-decode-execute loop that mimics a basic CPU’s operation. Initially, the compilation phase generates intermediate code stored in a dedicated memory area, after which the main function configures the VM environment by initializing essential registers including the program counter, stack pointer, and base pointer. The VM then enters an infinite loop in which it fetches the next instruction using the program counter, increments a cycle counter, and decodes the instruction through a series of conditional statements before executing it. For instance, when an immediate value opcode (IMM) is encountered, the value is loaded into an accumulator variable and the program counter is advanced. Control flow instructions, such as JMP and JSR, alter the execution sequence by updating the program counter, with JSR also saving the return address onto the stack to facilitate subroutine calls. Additionally, opcodes are mapped to a concatenated mnemonic string of 4-character identifiers, and in debug mode, these mnemonics and their operands are printed for clarity. Arithmetic and logical operations are handled in a stack-based manner using operations like ADD, SUB, MUL, DIV, and MOD. Furthermore, specialized opcodes manage stack frames (ENT and LEV), ensuring proper handling of function calls and local variables. Overall, it effectively bridges abstraction levels.

The **memory management** approach: How does the code handle memory allocation and deallocation?

The memory management strategy in the code effectively combines static memory pools with dynamic allocation to handle various runtime needs. At startup, the program allocates large, fixed-size memory blocks using malloc for different areas: the symbol table (sym) holds identifiers and metadata, the code/text area (e) stores the compiled instructions, the data area (data) reserves space for global variables and string literals, and the stack area (sp) supports function call management and local variable storage. These memory pools are all initialized to zero using memset, ensuring a clean state before the compilation begins. The design leverages static allocation to reduce overhead and fragmentation by dedicating ample space for compilation artifacts, while dynamic memory allocation is reserved for runtime needs. For

instance, when the VM encounters a MALC opcode, it calls the standard library malloc to allocate memory on the fly, and later frees it with the FREE opcode, thereby preventing memory leaks. Furthermore, operations like memset and memcmp are used to manage memory content during execution, supporting operations such as initializing buffers and comparing data blocks. The stack is managed by explicitly adjusting pointers for each function call. The ENT opcode saves the current base pointer and allocates space for new local variables, while the LEV opcode restores the previous execution context when a function returns. This dual approach of combining pre-allocated pools for predictable, high-volume data with dynamic allocation for flexible, runtime requirements results in a robust and efficient memory management system, ensuring that both static and dynamic data are handled optimally throughout the lifecycle of the program.