# Report on Key Algorithms and Concepts in the C4 Compiler

## 1. Lexical Analysis Process

The lexical analysis in C4 is performed by the `next()` function. Its primary role is to scan the raw source code character by character and convert it into a stream of tokens. Each token represents a syntactic element such as an identifier, a keyword (e.g., `if`, `while`), a numeric literal, or an operator (e.g., `+`, `-`, `*`).

- **Token Identification:**
  The function uses simple conditional checks to determine the type of token based on the current character. For example, if the character is a digit (`'0'–'9'`), it enters a loop to accumulate the number, converting it from a string to an integer. Similarly, alphabetic characters and underscores signal the start of an identifier.
- **Symbol Table and Hashing:**
  When an identifier is recognized, a hash is computed (using multiplication by 147 and a bit shift) to facilitate quick look-up in the symbol table. If the identifier exists, the token type is set accordingly; otherwise, the identifier is added to the table.
- **Handling Operators and Special Characters:**
  The function also manages operators, punctuation, and even multi-character operators (e.g., the conditional equality operator `==`). Comments and preprocessor directives (starting with `#`) are skipped. This simple yet efficient approach ensures that the source input is cleanly tokenized, forming the first phase of compilation.

## 2. Parsing Process

C4 uses a recursive descent parser implemented primarily in the `expr()` and `stmt()` functions. Although C4 does not construct a traditional Abstract Syntax Tree (AST) with node structures, it generates an equivalent representation by emitting intermediate code in the form of opcodes.

- **Expression Parsing (expr()):**
  The `expr()` function uses a technique known as "precedence climbing" (or top-down operator precedence). This approach allows the parser to correctly handle operator precedence and associativity without explicitly building an AST. As it processes tokens, it emits opcodes that represent operations (e.g., `ADD`, `SUB`, `MUL`) and immediate values. The result is a linear sequence of instructions that the virtual machine will execute.
- **Statement Parsing (stmt()):**
  The `stmt()` function deals with higher-level constructs such as control flow. It recognizes compound statements, conditionals (`if-else`), loops (`while`), and return statements. For control structures, the function emits jump and branch opcodes and later "patches" jump addresses to connect the different parts of the control flow. This flat representation of control flow is sufficient for a simple compiler like C4.

- **Absence of a Traditional AST:**
  Rather than building an explicit AST, the design of C4 favors direct code emission. This strategy minimizes memory usage and simplifies the compiler's structure, which is in keeping with the goal of implementing a self-hosting compiler in minimal code.

# 3. Virtual Machine Implementation

After parsing and code generation, the compiled program is executed by a simple virtual machine (VM) implemented in the `main()` function.

- **Opcode Execution Loop:**
  The VM uses a loop (`while (1)`) to fetch, decode, and execute opcodes from the generated code array. The program counter (`pc`) keeps track of the current instruction. Each opcode corresponds to an operation (for example, `IMM` loads an immediate value, `ADD` performs addition, and `JMP` transfers control).
- **Stack-Based Computation:**
  The VM is stack-based. Many instructions operate on values that are pushed onto or popped from the stack (`PSH`). This model simplifies arithmetic and control-flow operations because operands are implicitly managed through the stack.
- **Control Flow and Subroutine Handling:**
  Instructions such as `JMP`, `JSR` (jump to subroutine), and `LEV` (leave subroutine) are used to implement control flow, function calls, and returns. The VM uses a base pointer (`bp`) and a stack pointer (`sp`) to manage the call stack and local variable storage.
- **Debugging and System Calls:**
  Optional debugging output can be enabled to print the current cycle count and instruction, which is useful for understanding program execution. Additionally, system call opcodes (e.g., `OPEN`, `READ`, `PRTF`) allow the VM to perform I/O and memory management, integrating basic runtime functionality.

# 4. Memory Management Approach

Memory management in C4 is handled in a straightforward manner, emphasizing simplicity over sophistication.

- **Memory Pool Allocation:**
  At startup, the compiler allocates large fixed-size memory blocks (pools) for various purposes:
  - **Symbol Table:** Used for storing identifiers and keywords.
  - **Text Area:** Where the emitted intermediate code (opcodes and operands) is stored.
  - **Data Area:** Used for global variables and string literals.
  - **Stack:** Employed by the VM during program execution.
- **Dynamic Allocation and Deallocation:**
  The code uses `malloc()` for all memory allocations. Since C4 is a self-contained compiler with a short runtime, there is little need for complex garbage collection or

dynamic memory deallocation during execution. However, the virtual machine supports system call opcodes like `MALC` and `FREE` to handle dynamic memory requests made by the compiled program.

- **Minimal Overhead:**
  By allocating large contiguous memory pools, the design reduces fragmentation and simplifies pointer arithmetic throughout the compiler. The simplicity of this approach is appropriate for an educational tool designed to illustrate compiler construction.

# Conclusion

The C4 compiler exemplifies a minimal yet fully functional self-hosting C compiler. Its lexical analyzer efficiently tokenizes input using straightforward character-by-character scanning and hashing. The parser employs recursive descent with precedence climbing to directly emit a linear sequence of opcodes instead of building a traditional AST. The virtual machine executes these opcodes using a stack-based architecture, and memory management is kept simple with fixed-size pool allocations and basic dynamic allocation routines.

Together, these techniques demonstrate how fundamental compiler components can be implemented in a compact form, providing a solid foundation for understanding compiler design and implementation.