Key Algorithms in the C4 Compiler

1. **Lexical analysis process: How does the code identify and tokenize input?**

The lexical analysis is handled by the **next()** function, which scans the source code character by character and converts it into tokens.

Types of Tokens:
Keywords (if, while, return, int, etc.)
Identifiers (variable and function names)
Numbers (integer literals)
Operators (+, -, *, /, =, ==, etc.)

How Tokenization Works in this C4 code:

- The function reads characters from p (the source code pointer).
- It skips whitespace and comments.
- It detects keywords and identifiers using hash functions for faster lookup.
- It recognizes integer literals, supporting decimal, octal, and hexadecimal numbers.
- It handles string literals by storing their values in the data section.
- It returns the next token (tk), allowing the parser to process it.

Example:
If the input is:
if (x == 10) return x + 5;

The tokenized output is:
[**IF**] [**(**] [**ID:x**] [**==**] [**NUM:10**] [**)**] [**RETURN**] [**ID:x**] [**+**] [**NUM:5**] [**;**]

This tokenized representation is then passed to the parsing phase.

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

The parser processes tokens using functions like stmt() and expr(), which convert tokens into an abstract syntax tree (AST) or an equivalent structure for execution.

How Parsing Works in C4:

1. stmt() → Parses statements (e.g., if, while, return, {} blocks).
    - Uses recursion to process nested structures.
    - Identifies branches, loops, and function calls.
    - Generates bytecode instructions for execution.

2. expr() → Parses expressions and handles operator precedence.
    - Uses precedence climbing to evaluate expressions correctly.
    - Converts expressions into bytecode for the virtual machine.
    - Supports binary operations (+, -, *, /, &&, ||, ==, etc.).

Example of Parsing an if Statement
if (x > 5) return x * 2;

Parsing breakdown:

- stmt() detects an if statement.
- expr() processes the condition (x > 5).
- stmt() processes the return statement (return x * 2;).
- Bytecode is generated to evaluate x > 5 and conditionally execute return x * 2.

3. **The virtual machine implementation: How does the code execute the compiled instructions?**

The C4 compiler does not generate machine code directly. Instead, it produces bytecode that is executed by a virtual machine (VM).

How the Virtual Machine Works:

- The bytecode interpreter executes compiled instructions sequentially.
  Stack-based execution model:
    - Operands and results are pushed to the stack.
    - Instructions pop values, perform operations, and push results back.
    - Supports jump instructions (JMP, BZ, BNZ) for control flow.
    - Uses function calls (JSR) and stack adjustments (ADJ, LEV) for handling procedures.

Example Execution Process: For the code:
x = 5 + 10;

The generated bytecode instructions might look like:
IMM 5   // Load 5 onto the stack
IMM 10   // Load 10 onto the stack
ADD   // Pop two values, add them, push the result
SI   // Store the result in `x`

How it works:

- IMM loads constants onto the stack.
- ADD performs arithmetic.
- SI stores the result in memory.

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

The C4 compiler manages memory using both the stack and heap.

Stack Usage:

- Function calls push local variables onto the stack.
- The stack grows and shrinks automatically.
- Return addresses are stored to allow function execution.

Heap Usage:

- Dynamically allocated memory is handled using malloc().
- The symbol table, emitted code, and data storage are allocated on the heap.
- free() is used to manually release memory.
- The leaks tool ensures no memory leaks occur in C4.

Example of Memory Allocation in C4:
sym = malloc(poolsz); // Allocate memory for the symbol table
e = malloc(poolsz);   // Allocate memory for emitted bytecode
data = malloc(poolsz); // Allocate data storage
sp = malloc(poolsz);   // Simulated execution stack
Memory is properly managed, and no leaks were detected.

Conclusion

The C4 compiler implements a basic but efficient compiler architecture:

- Lexical analysis tokenizes source code (next()).
- Parsing builds a structured representation (stmt(), expr()).
- A virtual machine executes compiled instructions using a stack.
- Memory is efficiently managed between the stack and heap.