C4 Compiler: Key Algorithms and Concepts

March 03, 2025

1 Introduction

The C4 compiler, a minimalistic self-compiling C implementation by Robert Swierczek, demonstrates efficient design with four core functions: `next()`, `expr()`, `stmt()`, and `main()`. This report elucidates its key algorithms: lexical analysis, parsing, virtual machine execution, and memory management.

2 Lexical Analysis Process

The lexical analysis in C4 is handled by the `next()` function, which tokenizes the source code into a stream of tokens stored in global variables `tk` (token type) and `ival` (token value). The process operates as follows:

- **Input Traversal**: A global pointer `p` iterates through the source code, character by character, starting from `char *p`.
- **Token Identification**:
  - *Whitespace and Newlines*: Skips spaces and tracks line numbers via `line`, printing source if `src` is set.
  - *Identifiers*: Uses a simple hash function (`tk = tk * 147 + *p`) to recognize keywords and variables, storing them in the symbol table (`*sym`).
  - *Numbers*: Parses decimal, hexadecimal (e.g., `0xFF`), and octal literals, storing values in `ival`.
  - *Operators and Symbols*: Matches single- or multi-character operators (e.g., `==`, `++`) using nested conditionals.
  - *Strings and Characters*: Handles quotes, storing string data in `*data` and character values in `ival`.
- **Efficiency**: The single-pass approach avoids backtracking, leveraging global state to maintain context, making it lightweight yet effective for C4's subset of C.

3 Parsing Process

C4's parsing, split between `expr()` and `stmt()`, constructs an implicit abstract syntax tree (AST) directly through code emission, bypassing explicit tree construction for efficiency.

- **Expression Parsing (`expr(int lev)`)**: Uses a *precedence climbing* method to handle operator precedence:

- *Base Cases*: Processes literals (`Num`, `"string"`), variables, and unary operators (e.g., `*`, `-`), emitting opcodes like `IMM` and `LC`.
- *Binary Operators*: Iterates over tokens with precedence $\geq$ `lev`, emitting stack-based operations (e.g., `ADD`, `MUL`) while managing type (`ty`).
- *Control Flow*: Handles conditionals (`?:`) and short-circuit logic (`&&`, `||`) with jumps (`BZ`, `BNZ`).

- **Statement Parsing (`stmt()`)**: Recursively parses control structures:
  - *If/While*: Emits conditional jumps (`BZ`, `JMP`) and patches targets post-parsing.
  - *Return*: Emits `LEV` to exit functions.
  - *Blocks*: Processes sequences within {} recursively.
- **Implicit AST**: Rather than building a separate AST, C4 emits virtual machine instructions (stored in `*e`) during parsing, integrating code generation for compactness.

4 Virtual Machine Implementation

The virtual machine (VM) in `main()` executes the compiled instructions, providing a stack-based runtime environment:

- **Structure**: A single infinite loop fetches opcodes from `*pc` (program counter), incrementing a `cycle` counter for debugging.
- **Instruction Set**: Includes:
  - *Control*: `JMP`, `JSR`, `BZ`, `BNZ` for jumps and branches.
  - *Memory*: `LEA`, `LI`, `LC`, `SI`, `SC` for load/store operations.
  - *Arithmetic/Logic*: `ADD`, `MUL`, `AND`, etc., using stack operands.
  - *System Calls*: `PRTF`, `MALC`, etc., for I/O and memory.
- **Stack Management**: Uses `*sp` (stack pointer) and `*bp` (base pointer) to push/pop values and manage function frames via `ENT` and `LEV`.
- **Execution**: Each opcode updates the accumulator `a` or program flow, terminating with `EXIT`. Debug mode (`debug`) prints instructions.

5 Memory Management Approach

C4 employs a straightforward memory management strategy optimized for its minimalistic design:

- **Allocation**:
  - *Heap*: `main()` allocates fixed-size buffers (`poolsz = 256*1024`) for `sym`, `e`, `data`, and `sp` using `malloc`, totaling 1 MB.
  - *Data Segment*: `*data` stores globals and strings, incremented as needed with alignment.
  - *Runtime*: `MALC` and `FREE` opcodes allow dynamic allocation during execution.
- **Deallocation**: No explicit freeing of initial buffers occurs at program end, relying on OS cleanup. Runtime `free()` is supported via `FREE`.
- **Patterns**: Stack usage is minimal (few KB) for locals and calls, while heap dominates due to large static buffers, avoiding dynamic resizing for simplicity.

6 Conclusion

C4's algorithms reflect a balance of minimalism and functionality. Lexical analysis efficiently tokenizes input with a single pass, parsing emits code directly into a VM instruction stream, the VM executes with a simple stack model, and memory management prioritizes static allocation. This design enables self-compilation within a compact footprint.