# C to Lisp Converter Function Analysis

## Core Functions:

line-type: (defun line-type (line))

- ✓ Uses regular expressions to identify different types of C code constructs
- ✓ Contains multiple pattern matches for different code structures:
  - Control structures (if, for, while)
  - Variable operations (declarations, assignments)
  - Function-related lines (declarations, definitions)
  - Print statements (with variables or literals)
- ✓ Returns symbolic identifiers (e.g., 'if-statement, 'for-loop)
- ✓ Pattern matching is ordered from most specific to most general to ensure correct identification
- ✓ Handles edge cases like empty lines and unknown patterns
- ✓ Key to the entire conversion process as it determines how each line will be processed

conversion-foo: (defun conversion-foo (line-type))

- ✓ Acts as a function dispatcher using Common Lisp's function objects
- ✓ Maps each line type to its specific conversion function
- ✓ Uses cond statements for clear mapping of types to functions
- ✓ Returns actual function objects (not just names) using #' notation
- ✓ Enables dynamic selection of conversion strategies
- ✓ Provides extensibility for adding new conversion types
- ✓ Maintains separation of concerns between identification and conversion

convert: (defun convert (line))

- ✓ Central conversion orchestrator
- ✓ Two-step process:
  1. Determines line type using line-type function
  2. Retrieves and applies appropriate conversion function
- ✓ Creates a bridge between identification and conversion phases
- ✓ Handles all line types uniformly through a consistent interface
- ✓ Error handling is delegated to specific conversion functions
- ✓ Maintains code organization and clarity

convert-c-to-lisp-recursive: (defun convert-c-to-lisp-recursive (lines))

- ✓ Implements the main recursive conversion algorithm
- ✓ Special handling for consecutive variable operations:
  1. Groups related variable declarations/assignments
  2. Ensures proper scoping in the resulting Lisp code
- ✓ Maintains proper order of operations
- ✓ Handles nested structures and block scoping

- ✓ Uses tail recursion for efficiency
- ✓ Builds the final Lisp code structure incrementally
- ✓ Preserves program flow and semantic meaning
  Operation Converters

convert-logical-arithmetic-operation: (defun convert-logical-arithmetic-operation (expression))

- ✓ Transforms C-style infix operations to Lisp prefix notation
- ✓ Handles multiple operator types:
  1. Comparison operators (==, !=, <, >, <=, >=)
  2. Arithmetic operators (+, -, *, /)
- ✓ Converts C equality operators to Lisp equivalents:
  1. == becomes =
  2. != becomes /=
- ✓ Maintains operator precedence during conversion
- ✓ Handles both simple and complex expressions
- ✓ Preserves numerical and logical semantics

## Variable Handlers

convert-variable-declaration: (defun convert-variable-declaration (line))

- ✓ Processes standalone C variable declarations
- ✓ Extracts variable type and name using regex
- ✓ Converts to Lisp's setq format
- ✓ Initializes variables with appropriate default values
- ✓ Handles type information preservation where needed
- ✓ Generates appropriate Lisp variable bindings
- ✓ Includes error handling for malformed declarations

convert-variable-assignment: (defun convert-variable-assignment (line))

- ✓ Handles complex variable assignment operations
- ✓ Features:
  - Proper indentation maintenance
  - Conversion of infix operations to prefix
  - Special handling for comparison operators
  - Support for compound assignments
- ✓ Multiple assignment patterns:
  - Simple value assignments
  - Arithmetic operations
  - Logical operations
- ✓ Preserves operation semantics while converting syntax
- ✓ Includes detailed error handling and formatting

convert-variable-by-function: (defun convert-variable-by-function (line))

- ✓ Specializes in function call assignments
- ✓ Processes:
  - Function name extraction
  - Argument list parsing
  - Return value assignment
- ✓ Handles multiple argument cases
- ✓ Maintains function call semantics
- ✓ Preserves argument order and types
- ✓ Formats output for readability

convert-variable-assignments: (defun convert-variable-assignments (lines))

- ✓ Manages groups of related variable operations
- ✓ Complex decision making:
  - When to use let vs. setq
  - How to group related assignments
  - Proper scoping rules
- ✓ Handles multiple consecutive assignments
- ✓ Maintains variable initialization order
- ✓ Creates appropriate binding structures
- ✓ Ensures proper variable scope and visibility

## Control Structure Converters

convert-if-statement: (defun convert-if-statement (line))

- ✓ Transforms C if statements into Lisp conditional expressions
- ✓ Features:
  - Condition extraction and conversion
  - Block structure creation
  - Proper indentation management
  - Progn insertion for multiple expressions
- ✓ Maintains program flow semantics
- ✓ Handles complex conditions
- ✓ Manages state for nested if statements
- ✓ Preserves logical structure of conditions

convert-for-loop: (defun convert-for-loop (line))

- ✓ Converts C-style for loops to Lisp loop constructs
- ✓ Complex processing:
  - Initialization extraction
  - Condition parsing

    - Increment/decrement handling
- ✓ Determines appropriate loop type
- ✓ Handles different loop patterns:
    - Counting up/down
    - Step sizes
    - Complex conditions
- ✓ Maintains loop semantics and efficiency

convert-while-loop: (defun convert-while-loop (line))

- ✓ Transforms C while loops into Lisp loop structures
- ✓ Features:
    - Condition extraction and conversion
    - Loop body structure creation
    - Proper indentation handling
- ✓ Maintains loop invariants
- ✓ Preserves condition semantics
- ✓ Handles complex loop conditions
- ✓ Ensures proper loop termination logic

## Function Handlers

convert-function-declaration: (defun convert-function-declaration (line))

- ✓ Processes C function declarations into Lisp ftype declarations
- ✓ Handles:
    - Return type conversion
    - Parameter type conversion
    - Function signature preservation
- ✓ Type system mapping:
    - C types to Lisp types
    - Parameter type lists
    - Return type specifications
- ✓ Generates proper type declarations
- ✓ Maintains function interface specifications

convert-function-definition: (defun convert-function-definition (line))

- ✓ Converts C function definitions to Lisp defun forms
- ✓ Complex processing:
    - Parameter list extraction
    - Return type handling
    - Function body preparation
- ✓ Maintains function semantics
- ✓ Handles different function patterns:

    - Void functions
    - Return value functions
    - Parameter variations
- ✓ Sets up proper function environment

## Output Handlers

convert-print-function-variable: (defun convert-print-function-variable (line))

- ✓ Converts printf with variables to Lisp format
- ✓ Handles:
    - Format string conversion
    - Variable argument processing
    - Format specifier translation
- ✓ Maintains output formatting
- ✓ Preserves string literals
- ✓ Handles multiple variables
- ✓ Manages format string escape sequences

convert-print-function-literal: (defun convert-print-function-literal (line))

- ✓ Simpler version of variable printf converter
- ✓ Processes:
    - String literal extraction
    - Format specifier conversion
    - Proper quotation
- ✓ Maintains string formatting
- ✓ Handles escape sequences
- ✓ Preserves output appearance

## Utility Functions

replace-percent: (defun replace-percent (input-string))

- ✓ Converts C format specifiers to Lisp equivalents
- ✓ Character-by-character processing
- ✓ Handles:
    - % to ~ conversion
    - String manipulation
    - Format preservation
- ✓ Maintains string integrity
- ✓ Preserves non-format characters

read-file-recursive: (defun read-file-recursive (stream &optional acc))

- ✓ Implements efficient file reading
- ✓ Features:
  - Recursive processing
  - Line order preservation
  - Memory efficient
- ✓ Handles:
  - File streaming
  - Line accumulation
  - EOF conditions
- ✓ Maintains file structure

write-file: (defun write-file (file content))

- ✓ Manages output file generation
- ✓ Handles:
  - File creation/overwriting
  - Content formatting
  - Line writing
- ✓ Maintains:
  - File permissions
  - Output formatting
  - Error handling

## Special Case Handlers

convert-return-value: (defun convert-return-value (line))

- ✓ Processes C return statements
- ✓ Handles:
  - Simple returns
  - Complex expressions
  - Infix to prefix conversion
- ✓ Maintains return semantics
- ✓ Preserves expression evaluation
- ✓ Handles multiple return patterns

convert-close-bracket: (defun convert-close-bracket (line))

- ✓ Manages C block endings
- ✓ Features:
  - Block level tracking
  - Proper closure generation
  - Indentation management

- ✓ Maintains block structure
- ✓ Handles nested blocks
- ✓ Preserves code organization

convert-empty-line: (defun convert-empty-line (line))

- ✓ Preserves code formatting
- ✓ Maintains:
  - Visual separation
  - Code readability
  - Document structure
- ✓ Handles whitespace preservation
- ✓ Maintains code layout

convert-unknown: (defun convert-unknown (line))

- ✓ Fallback handler for unrecognized patterns
- ✓ Features:
  - Error preservation
  - Debugging support
  - Code documentation
- ✓ Maintains:
  - Original line as comment
  - Conversion traceability
  - Error tracking capability