

# Mumps Programming Language Interpreter & Compiler AI Summary and Overview by Google Gemini

Kevin C. O'Kane

kc.okane@gmail.com  
okane@uni.edu

<http://threadsafebooks.com/>  
<http://www.cs.uni.edu/~okane/>

March 30, 2026

The following is the Google Gemini 0.34.0 summary and analysis of the code as of March 27, 2026. It presents an interesting and useful guide to the structure and layout of the source code which may be useful to developers. The initial prompts asked Gemini to summarize the code.

---

I've completed the analysis of `interp-interp.cpp`. This file implements the core of a Mumps interpreter, with the `Interpret` function serving as the central execution loop. It handles standard Mumps commands (`SET`, `GOTO`, `DO`, `IF`, `FOR`, `READ`, `WRITE`, etc.) using a large switch statement and manages execution state through a gosub stack and a MSV (Mumps State Vector) structure.

## Key findings:

- Command Dispatch: Commands are identified by their first letter (case-insensitive) and dispatched to specific handlers.
  - Execution Flow: It supports complex flow control, including nested `DO` and `FOR` loops, as well as `XECUTE` for dynamic code execution.
  - Data Management: Local variables are managed via `sym_`, and global arrays (persistent storage) are handled through `Mglobal`.
  - Expression Evaluation: The interpreter relies heavily on an external `parse_` function (likely in `interp-parse.cpp`) for evaluating expressions and handling indirection.
  - Extended Features: It includes non-standard support for `SQL` and `SHELL` commands directly within the Mumps environment.
  - Error Handling: Comprehensive error trapping is implemented, with support for user-defined error traps and standard Mumps error codes.
- ✦ I will continue the analysis by examining `interp-parse.cpp`, as it handles the expression evaluation and indirection logic that the `Interpret` function relies on.
- ✓ ReadFile `interp-parse.cpp`
- ✦ I've analyzed `interp-parse.cpp`, which complements `interp-interp.cpp` by providing expression parsing and evaluation.

Key findings from `interp-parse.cpp`:

1. `parse_function`: The core of this file. It uses a recursive-descent style (partially manual with stacks) to evaluate Mumps expressions. It handles:
  - \* Operators: Supports a wide range of Mumps operators including arithmetic (+, -, \*, /, \, \*\*, #), logical (&, !), concatenation (\_), and comparisons (=, <, >, ], [[, etc.).
  - \* Operands: Handles numeric literals, string literals (with "" escaping), and variable names.
  - \* Variables: Resolves local variables via `sym_` and global variables via `Mglobal`. It also handles built-in variables (\$) and functions.
  - \* Indirection: Implements Mumps indirection (@) which dynamically re-evaluates strings as expressions.
  - \* Parentheses: Manages operator precedence and nested expressions.
2. Pattern Matching: Implements the Mumps pattern match operator (?).
  - \* `match()`: Core pattern matching logic for individual pattern codes (C, N, P, A, L, U, E).
  - \* `patrn()` and `patrn1()`: Handle the parsing and execution of pattern match expressions, supporting counts and ranges (e.g., 1N, .2A).
3. Utility Functions:
  - \* `cannon()`: Converts numeric strings to canonical Mumps format (e.g., removing leading/trailing zeros).
  - \* `getstr2()`: Reads a line from an input unit.
  - \* `readf_()`: Reads an entire Mumps source file into the interpreter's buffer.
  - \* `inout()`: Handles output operations like tab (?), new line (!), and page (#).
  - \* `prnterr()`: A centralized error reporting function that translates internal error codes into human-readable Mumps error messages and provides context (line number and source text).
4. Integration with Interpreter State: It heavily uses the MSV (Mumps State Vector) to maintain the execution context, symbol table access, and I/O state.

Summary of the Interpreter Architecture (so far):

- \* `interp-interp.cpp`: The execution engine (the "brain") that steps through commands.
- \* `interp-parse.cpp`: The expression evaluator (the "logic") that understands the data and calculations within those commands.

Next, I'll examine `interp-sym.cpp` to understand how the local symbol table is managed, as both files rely on the function.

✦ I will now read `interp-sym.cpp` to understand how the local symbol table and variable management are implemented.

✓ ReadFile `interp-sym.cpp`

✦ I've analyzed `interp-sym.cpp`, which manages the local symbol table for the Mumps interpreter.

Key findings from `interp-sym.cpp`:

#### 1. Symbol Table Structure:

- \* The symbol table is implemented as an array of linked lists (`svPtr->start[SYM_MAX]`), where each element in the array represents a "nesting level" or "scope" (used for DO calls and NEW commands).

- \* Each node in the linked list is a struct stab, containing:
  - \* name: The Mumps variable name (processed for canonical format).
  - \* data: A pointer to a C++ std::string containing the variable's value.
  - \* exec: A flag indicating if the variable was stored at the current level only (for parameters).
  - \* OrigName: Used for call-by-reference logic.
  - \* next: Pointer to the next variable in the list at the same level.
- 2. The `sym_` function: The core interface for symbol table operations, handling a wide range of tasks based on `symflg`:
  - \* `SymStore (0, 100, 200)`: Stores or creates a variable. Level-specific storage (100, 200) is used for subroutine parameters and call-by-reference.
  - \* `SymRetrieve (1, 11)`: Retrieves a variable's value by searching from the current nesting level down to level 0.
  - \* `SymKill (2, 4, 5)`: Deletes specific variables, all variables, or all except a list of specified variables.
  - \* `SymData (6)`: Implements the Mumps `$DATA()` function logic.
  - \* `SymNew (7, 8, 99)`: Implements Mumps `NEW` logic, creating a new scope level and managing variables that should be preserved.
  - \* `SymOrder (33)` and `SymNext (3)`: Implement `$ORDER()` and `$NEXT()` logic for local arrays.
- 3. Scope Management:
  - \* `_SymFree()`: Pops the current symbol table level, cleaning up memory and handling call-by-reference updates when returning from a subroutine.
  - \* Levels are incremented (`svPtr->_Sym++`) during subroutine calls and `NEW` operations, and decremented during `QUIT` or end of scope.
- 4. Utility & Extended Functions:
  - \* `BuildLocal()`: Manages a stack for building local array references.
  - \* `SysDump()` and `SysLoad()`: Provide mechanisms to save and restore the local symbol table to/from disk (likely for `JOB` command or persistence).
  - \* `GlobalQuery()`, `Qlength()`, `Qsub()`: Helper functions for global/local array introspection (`$QUERY`, `$QLENGTH`, `$QSUBSCRIPT`).
  - \* `Eval()`: A convenience wrapper around the parser for evaluating expressions within the symbol management ontext.
  - \* Implementations of Mumps built-in functions: `_ascii`, `_extract`, `_find`, `_horolog`, `_justify`, `_length`, `_random`.
- 5. Canonical Formatting:
  - \* The code uses `keyfix` and manual bit manipulation (`*kp & 0x80 ? 1 : *kp`) to normalize variable names and array subscripts, ensuring consistent lookup.

#### Summary of the Component:

`interp-sym.cpp` provides a robust, multi-level symbol table implementation that correctly handles Mumps' unique scoping rules, array structures, and dynamic variable creation. It bridges the gap between the high-level interpreter/parser and the low-level C++ string and memory management.

Next, I'll examine `interp-fcn.cpp` to see how the remaining built-in Mumps functions are

implemented. After that, I'll look at the global array interface, likely in `interp-lobalb.cpp` or similar.

✦ I will now read `interp-fcn.cpp` to examine the implementation of Mumps built-in functions.

✓ ReadFile `interp-fcn.cpp`

✦ I've analyzed `interp-fcn.cpp`, which implements the built-in Mumps functions for the interpreter.

Key findings from `interp-fcn.cpp`:

#### 1. User-Defined Functions (`$$FUNCTION`):

- \* Handles calls to extrinsic functions (subroutines that return a value).
- \* Supports both local and external file references (e.g., `$$label^file(args)`).
- \* It manages a nested `Interpret` call by creating a new MSV (Mumps State Vector) and copying the symbol table pointers, allowing for recursion and proper scoping.

#### 2. Built-in Function Dispatch (`fcn` function):

- \* Uses an opcode table to map the second character of the function name (lowercase) to a switch case.
- \* Handles argument parsing, including coded delimiters (206 for open paren, 207 for close, 208 for comma).

#### 3. Key Function Implementations:

- \* `$EXTRACT()`: Supports both RHS (retrieval) and LHS (assignment to a substring). LHS `$E()` is complex, as it involves fetching the current value (local or global), modifying it, and storing it back.
- \* `$PIECE()`: Delegated to `_piece` (likely in `interp-piece.cpp`). Handles both RHS and LHS.
- \* `$SELECT()`: Evaluates a series of condition:expression pairs and returns the first expression where the condition is true.
- \* `$FIND()`: Standard string search.
- \* `$FNUMBER()`: Advanced number formatting using `strfmon`, supporting Mumps-specific flags like +, -, ,, T, P.
- \* `$JUSTIFY()`: Handles string padding and decimal rounding.
- \* `$TRANSLATE()`: Character-for-character replacement or deletion.
- \* `$HOROLOGY()`: Returns the current Mumps system time and date.
- \* `$DATA()`: Handled via `sym_` (local) or `Mglobal` (global).
- \* `$GET()`: Returns the value of a variable or a default if it's undefined.
- \* `$ORDER()` / `$NEXT()`: Iterates through array subscripts.
- \* `$NAME()`, `$QUERY()`, `$QLength()`, `$QSUBSCRIPT()`: Tools for meta-programming and array introspection.

#### 4. Special Variables:

- \* `$X` and `$Y`: Current horizontal and vertical cursor positions on the output device.
- \* `$IO`: Current I/O unit number.
- \* `$TEST`: Result of the last `IF` command or command with a timeout.
- \* `$JOB`: Current process ID (returned in hex).

- \* \$STORAGE: Remaining space in the user partition.

## 5. Z-Variables:

- \* Supports a few custom variables like \$ZTABLE, \$ZTABSIZ, and \$ZSQLOUTPUT for RDBMS integration.

### Summary of the Component:

interp-fcn.cpp is a dense implementation of the Mumps standard library. It showcases the language's strong support for string manipulation, sparse arrays, and dynamic execution. The implementation of LHS \$EXTRACT and \$PIECE is particularly noteworthy as it requires tight integration with the symbol table and global storage systems.

Next, I'll examine interp-piece.cpp to see the implementation of the \$PIECE function, and then I'll move to the global array interface.

✦ I've analyzed interp-piece.cpp, which implements the \$PIECE function.

### Key findings from interp-piece.cpp:

#### 1. Functionality: \$PIECE is a fundamental Mumps function used to extract or replace substrings delimited by a specified string (the "delimiter").

- \* in: The source string.

- \* key: The delimiter.

- \* start and end: The 1-based index of the piece(s) to operate on.

- \* setpiece: A boolean flag; 0 for retrieval (RHS), 1 for assignment (LHS).

- \* lhs: The value to insert when setpiece is 1.

- \* out: The variable name to store the result when setpiece is 1, or the buffer for the result when setpiece is 0.

#### 2. Retrieval (RHS):

- \* The function calculates the start and end positions by searching for occurrences of the delimiter using xindex.

- \* It extracts the specified piece(s) (including intermediate delimiters if end > start) and returns them in out.

- \* If the requested piece doesn't exist, it returns an empty string.

#### 3. Assignment (LHS):

- \* This is more complex. It reconstructs the entire variable value.

- \* It takes the part of the string before the target piece, appends the new value (lhs), and then appends the part of the string after the target piece.

- \* It then stores the updated string back into either a local variable (via sym\_) or a global array (via Mglobal), depending on the variable name in out.

- \* Limitation/Observation: The implementation for LHS seems to simplify the concatenation and doesn't explicitly handle cases where the original string needs to be extended with delimiters if the target piece index is beyond the current length (a standard Mumps feature). It mostly focuses on replacing an existing piece.

#### 4. Error Handling: Basic checks for negative indices or end < start are performed, resulting in an error message and a signal to terminate.

### Summary of the Component:

interp-piece.cpp provides the core logic for one of Mumps' most powerful string processing

tools. It integrates directly with the symbol table and global storage systems, enabling efficient data manipulation within the interpreter.

Next, I'll examine the global array interface. I see several candidates: `interp-globalb.cpp` (and its `.in` version) and `interp-native-btree.cpp`. Since `SQLITE` is defined in most files, I suspect the global storage is backed by `SQLite`.

✓ ReadFile `interp-globalb.cpp`

✦ I've analyzed `interp-globalb.cpp`, which acts as the high-level interface for Mumps global arrays.

Key findings from `interp-globalb.cpp`:

#### 1. Interface Function `Mglobal`:

- \* This is the primary function called by the interpreter (`Interpret`) and parser (`parse_`) to interact with persistent global arrays.
- \* It takes an operation code `g` (e.g., `RETRIEVE`, `STORE`, `NEXT`, `GKILL`), a key (the array reference), and a data buffer `bd`.

#### 2. Database Backend (`SQLite` vs. `Native`):

- \* The file is heavily conditionally compiled using `#define SQLITE`.
- \* When `SQLITE` is defined:
  - \* Operations are mostly direct passes to the `btree()` function (likely implemented in another file like `interp-sql.cpp` or similar, despite the "btree" name which might be a legacy abstraction).
  - \* `keyfix(key)` is called before most operations to normalize the Mumps global reference into a canonical format suitable for the database.
- \* When `SQLITE` is not defined (`Native` section):
  - \* It implements complex logic for `$NEXT` and `$ORDER` operations by manipulating the key strings (e.g., adding `~` or `\x02` to find the "next" or "previous" key in a B-Tree).
  - \* It handles unpadding and canonicalization of keys returned from the native B-Tree.

#### 3. Key Operations:

- \* `RETRIEVE`: Fetches the value associated with a global array node.
- \* `STORE`: Sets the value for a node.
- \* `GKILL`: Deletes a node (and potentially its sub-nodes, though recursive kill is often handled at a higher level or within the database logic).
- \* `NEXT`, `ORDERNEXT`, `ORDERPREV`: Navigate through the sparse array subscripts.

#### 4. Legacy and Portability:

- \* Comments indicate some code was originally in Fortran, explaining certain array indexing conventions.
- \* The use of `keyfix` is consistent across the interpreter, ensuring that a global like `^A(1)` is always stored and retrieved the same way, regardless of how the user types the subscripts.

Summary of the Component:

`interp-globalb.cpp` provides a clean abstraction layer for Mumps global arrays. It separates the interpreter's logic from the physical storage implementation, allowing the system to switch between a native B-Tree and a relational database like `SQLite` while maintaining

standard Mumps behavior.

Since SQLITE is the active path, the next logical step is to examine `interp-sql.cpp` (or `interp-native-btree.cpp` if I want to see the alternative) to see the actual database implementation. Given the file list, `interp-sql.cpp` seems more relevant to the current configuration.

- ✦ I've analyzed `interp-sql.cpp`, which implements the SQLite backend for Mumps global arrays.

Key findings from `interp-sql.cpp`:

#### 1. Architecture:

- \* Mumps global arrays are mapped to a single SQLite table named `mumps`.
- \* This table has a fixed number of columns (configured as 14 in this version, with up to 32 supported).
- \* Columns `a1` through `a13` (approx.) store the subscripts (indices) of the global array.
- \* Column `a14` (approx.) stores the actual data value associated with the node.
- \* This structure allows for up to 13 levels of nesting in global arrays.

#### 2. Mapping Mumps to SQL:

- \* **RETRIEVE**: Translates to a `SELECT a14 FROM mumps WHERE a1='...' AND a2='...' ... LIMIT 1`. It pads unused subscript columns with empty strings ('') to ensure an exact match.
- \* **STORE**: First performs a `DELETE` for the existing key to handle updates/replacements, and then an `INSERT INTO mumps VALUES (...)`.
- \* **GKILL**: Deletes rows. If a partial key is provided (e.g., `^A`), it deletes all rows where `a1='A'`. If `^A(1)` is provided, it deletes rows where `a1='A' AND a2='1'`. This effectively implements Mumps' hierarchical kill.
- \* **ORDERNEXT / ORDERPREV**: Uses `SELECT aX FROM mumps WHERE ... AND aX > '...' ORDER BY aX ASC LIMIT 1` (or `DESC` for previous). This leverages SQLite's indexing for efficient navigation.
- \* **XNEXT** (for `$QUERY`): Uses a more complex "tuple comparison" style query (e.g., `(a2, a3, a4) > ('1', '2', '3')`) to find the next node in the entire global tree.

#### 3. SQLite Integration:

- \* Uses the `sqlite3_exec` interface with a callback function `rslt_callback`.
- \* Database file is expected to be `mumps.sqlite` in the current directory.
- \* Implements a busy handler (`sqlite3_busy_timeout`) to manage concurrent access.
- \* Disables the journal (`pragma journal_mode = OFF`) for performance, though this sacrifices some crash safety.

#### 4. Data Integrity and Formatting:

- \* `lcheck` and `dcheck`: Verify that index and data lengths don't exceed configured maximums.
- \* `embed`: Escapes single quotes (') by doubling them, preventing SQL injection and ensuring correct data handling.
- \* `keyfix`: Used (at the Mglobal level in `interp-globalb.cpp`) to ensure keys are in canonical format before reaching this layer.

#### 5. `rslt_callback`:

- \* Converts SQL row results back into Mumps format. For multi-column results (like from

\$QUERY), it reconstructs the Mumps global reference string (e.g., ^A("sub1","sub2")).

#### Summary of the Component:

interp-sql.cpp is a clever mapping of Mumps' hierarchical, sparse array model onto a relational database. By flattening the hierarchy into fixed columns, it achieves high performance for deep trees while maintaining the semantic behavior expected by Mumps programmers.

I have now covered the major components of the interpreter:

- \* Execution Loop: interp-interp.cpp
- \* Expression Parsing: interp-parse.cpp
- \* Local Variables: interp-sym.cpp
- \* Built-in Functions: interp-fcn.cpp
- \* String Tools: interp-piece.cpp
- \* Global Arrays (Abstraction): interp-globalb.cpp
- \* Global Arrays (Storage): interp-sql.cpp

Other files like interp-arith.cpp (arithmetic), interp-fnumber.cpp (formatting), and interp-zfcns.cpp (custom functions) likely contain additional specialized logic.

- ✦ I will now read interp-arith.cpp, interp-fnumber.cpp, and interp-zfcns.cpp to understand how arithmetic, number formatting, and custom functions are implemented in the Mumps interpreter.

✓ ReadFile interp-arith.cpp

- ✦ I've analyzed interp-arith.cpp, which implements arithmetic operations for the Mumps interpreter.

Key findings from interp-arith.cpp:

1. String-Based Arithmetic: As is standard for Mumps, all arithmetic operations take string operands and produce string results. This allows the language to handle numbers of varying precision and format seamlessly.

#### 2. Flexible Precision Support:

- \* The file uses extensive conditional compilation (#ifndef MULTI\_PRECISION, #ifdef BIGFLOAT) to support different numeric backends:

- \* Standard double: Default floating-point precision.
- \* long double (BIGFLOAT): Extended precision floating point.
- \* GMP/MPFR (MULTI\_PRECISION): Arbitrary-precision integers and floating-point numbers. This is a powerful feature not always found in basic Mumps implementations.

#### 3. Arithmetic Operators:

- \* Implements add, sub, mult, divx (division), divi (integer division), modulo, and expx (exponentiation).
- \* Many of these use a common template file mumpsc/numSetup.h via the NS\_... opcodes, likely to handle the boilerplate of string-to-numeric conversion and back.

#### 4. Mathematical Functions:

- \* Provides a full suite of transcendental and utility functions: log, log2, log10, sin, cos, tan, asin, acos, atan, sqrt, exp, abs, trunc, floor, ceil.



- \* Like the operators, these are implemented for both standard and multi-precision backends.

## 5. Comparison and Utility:

- \* numcomp: The core numeric comparison function. It correctly handles comparing two numeric strings by converting them to their numeric equivalents first.
- \* Mltoa: A custom "long to string" conversion function, likely optimized for Mumps' needs.
- \* floatstring: A helper to detect if a string represents a floating-point number (contains a . or e).
- \* mps\_justify: Implements the numeric formatting logic for the \$JUSTIFY function.

## 6. Error Handling:

- \* Includes checks for integer overflow and division by zero (especially in the multi-precision paths).
- \* Uses a custom exception mechanism (throw MumpsGlobalException()) for certain numeric errors.

### Summary of the Component:

interp-arith.cpp is a comprehensive numeric library that gives the Mumps interpreter its mathematical capabilities. The support for arbitrary-precision arithmetic via GMP/MPFR is a standout feature, making this implementation suitable for scientific or high-precision financial applications.

Next, I'll read interp-fnumber.cpp. Since I already saw \$FNUMBER logic in interp-fcn.cpp, this file likely contains the detailed implementation of that function or related formatting tools.

Then I'll finish with interp-zfcns.cpp.

- ✓ ReadFile interp-fnumber.cpp
- ✓ ReadFile interp-zfcns.cpp
- ✓ ReadFile interp-fnumber.cpp
- ✦ I've analyzed interp-fnumber.cpp and interp-zfcns.cpp.

### Key findings from interp-fnumber.cpp:

This file is a specialized wrapper for the \$FNUMBER() function. It provides the \_fnumber function, which:

- \* Takes numeric value and formatting arguments.
- \* Constructs a Mumps-style internal call string (e.g., \$fn(value,arg1,arg2)).
- \* Delegates the actual work to the fcn() function (in interp-fcn.cpp), which we already saw uses strfmon for localized currency and number formatting.
- \* This wrapper is likely used by the compiler's code generation or for internal interpreter calls where a more C-like interface is needed.

### Key findings from interp-zfcns.cpp:

This is a massive file implementing a wide range of non-standard "Z" functions, significantly extending the Mumps language.

## 1. GUI Integration (GTK):

- \* Extensive support for creating and managing GTK widgets directly from Mumps.
- \* Includes functions for: Dialogs, TreeViews (with GtkTreeLevelAdd), SpinButtons, Entries, Labels, Calendars, FileChoosers, ColorChoosers, and Switches.

- \* This allows Mumps programs to have modern Linux graphical user interfaces.

## 2. Scientific and Bioinformatics Tools:

- \* Sequence Alignment: `zsmithwaterman` implements the Smith-Waterman algorithm for local sequence alignment.
- \* Text Processing & IR: `zstem` (suffix removal), `zstoplookup` (stopword removal), `zsynlookup` (synonym handling), `zshred` (k-mer generation), `zbgmsearch` (Boyer-Moore-Gosper string search).
- \* Similarity Metrics: `zzjaccard`, `zzcosine`, `zzsiml` (Dice/similarity), `zzsoundex`.
- \* Vector/Matrix Ops: `zzavg`, `zzmax`, `zzmin`, `zzsum`, `zzmultiply` (matrix multiply), `zztranspose`, `zzcentroid`, `zzdoccorrelate`.

## 3. Database and SQL Management:

- \* Functions to query database configuration: `zindexsize`, `zdatasize`, `zdbname`, `zdbfile`, `zindexmax`.
- \* Transaction Control: `zsqlite` handles `BEGIN TRANSACTION`, `COMMIT`, `ROLLBACK`, `SAVEPOINT`, and `PRAGMA`.
- \* `zsqlOpen`: Checks if the SQL connection is active.
- \* `zdatabase`: Returns the type of database backend (SQL vs. Native).

## 4. System and File Utilities:

- \* `zsystem`: Executes shell commands.
- \* `zfiletest`: Tests file properties (using shell if [ ... ]).
- \* `zbasename`: Returns the base name of a path.
- \* `zchdir`: Changes the current working directory.
- \* `ztell` / `zseek`: File pointer manipulation.
- \* `zgetenv`: Accesses environment variables.
- \* `zdump` / `zrestore`: Native B-Tree backup/restore (disabled in SQL mode).

## 5. String and Numeric Utilities:

- \* `zlower`: Case conversion.
- \* `znormal` / `zalter`: Text normalization (removing punctuation, lowercase).
- \* `znoblanks` / `zblanks`: Space removal/normalization.
- \* `znumeric`: Checks if a string is a valid number.
- \* `zhtml`: URL/HTML encoding.
- \* Extended Math: `zsqrt`, `zsin`, `zcos`, `ztan`, `zasin`, `zacos`, `zatan`, `zexp`, `zlog`, `zpow`, `zceil`, `zfloor`, `ztrunc`.

## Summary of the Component:

`interp-zfcns.cpp` transforms this Mumps implementation from a legacy database language into a versatile platform for GUI development, bioinformatics, and information retrieval. The inclusion of Smith-Waterman and Boyer-Moore-Gosper algorithms alongside GTK bindings is quite unique.

I have now completed a thorough analysis of the Interpreter components. The system is a hybrid: a classic Mumps core with a modern SQLite backend and extensive C++-powered extensions for specialized domains.

Would you like to move on to the Compiler components (files like compiler-mumps.c, compiler-parse.c, etc.)? These likely translate Mumps source into C code for compilation.

- ✦ I will begin the analysis of the compiler components by listing the relevant compiler-\*.c files to get an overview of the compiler's structure.

✓ FindFiles 'compiler-\*.c'  
Found 21 matching file(s)

- ✦ I'll start by reading compiler-mumps.c to understand the overall compilation process and how it interacts with the other files.
- ✓ ReadFile compiler-mumps.c
- ✦ I've analyzed compiler-mumps.c, which is the main driver for the Mumps-to-C++ compiler.

Key findings from compiler-mumps.c:

1. Overview: The compiler translates Mumps source code into C++ source code. This C++ code is then compiled and linked with the runtime library (which I analyzed earlier) to create an executable.
2. Output Generation:
  - \* It generates a .m file (which is actually C++ code).
  - \* It writes necessary #include directives and external function declarations to the output file.
  - \* It includes support for standard Mumps features and the custom "Z" functions and GTK interface.
3. Compilation Loop (next\_line):
  - \* The compiler reads the Mumps source file line by line.
  - \* Labels: It identifies labels (which must start at the beginning of a line) and generates C++ labels or functions as appropriate.
  - \* Subroutines/Functions: It handles Mumps entry points (marked with ^ or ^^) by generating C++ functions. It supports parameter passing and call-by-reference.
  - \* Indentation (Doty Blocks): It handles Mumps block structure (lines starting with .) by generating do { ... } while(0) blocks and using setjmp/longjmp for flow control within blocks.
  - \* Command Dispatch: For each command on a line, it identifies the command (e.g., SET, IF, DO, READ, WRITE) and calls a specific handler function (e.g., set\_code, if\_code, do\_code).
4. Special Features:
  - \* Inline C/C++: Mumps lines starting with % or + are emitted directly as C++ code.
  - \* SQL/Shell: It has built-in support for SQL and SHELL commands, which it translates into calls to the interpreter's Interpret function.
  - \* HTML Support: The HALT (or HT) command can be used for HTML output generation, escaping special characters appropriately.
  - \* Try/Catch: It supports Mumps TRY and CATCH blocks, translating them into C++ try/catch.
5. Integration with Interpreter:

- \* The compiled code still uses the MSV (Mumps State Vector) to maintain state.
- \* It uses the same symbol table (sym\_) and global array (Mglobal) functions as the interpreter.
- \* It can call back into the interpreter using the Interpret function for dynamic code execution (XECUTE, indirection).

## 6. Code Generation Strategy:

- \* It makes heavy use of setjmp and longjmp to implement Mumps' unique control flow, especially for QUIT from nested blocks and GOTO out of subroutines.
- \* It generates explicit C++ code for variable assignments, comparisons, and I/O.

## Summary of the Compiler Architecture:

The compiler is a single-pass translator. It uses a series of modular handlers for different Mumps commands, while the main loop manages the source file and overall program structure. By translating to C++, it leverages the C++ compiler's optimization while providing a runtime environment that mimics the Mumps language's behavior.

Next, I'll examine compiler-parse.c to see how expressions are parsed during compilation, and then I'll look at some of the command-specific handlers like compiler-set.c or compiler-if.c.

### ✓ ReadFile compiler-parse.c

- ✦ I've analyzed compiler-parse.c, which implements the expression parser for the compiler.

## Key findings from compiler-parse.c:

1. Code Generation Strategy: Unlike the interpreter's parse\_ function (which evaluates expressions immediately), the compiler's parse\_ function generates C++ code to evaluate the expression at runtime. It uses a stack-based approach to handle operator precedence and nested expressions.
2. Operator Translation:
  - \* Mumps operators are translated into C++ function calls or logic.
  - \* Arithmetic (+, -, \*, /, \) becomes calls to add, sub, mult, divx, divi.
  - \* Comparisons (=, <, >, ]) become calls to strcmp or numcomp.
  - \* Logical operators (&, !) are translated into C++ && and || with numeric conversion via numcomp.
  - \* Pattern matching (?) becomes a call to the pm function.
  - \* Indirection (@) generates code that calls the interpreter's parse\_ function at runtime for dynamic evaluation.
3. Variable and Function Handling:
  - \* Local Variables: It generates code to retrieve variables via sym\_ or direct C++ variable access if they are declared.
  - \* Global Variables: It generates code that builds a canonical global reference string and then calls Mglobal (wrapped in #G#...# markers for later processing).
  - \* Built-in Functions: It has specialized Expand... functions (like ExpandOrder, ExpandPiece, ExpandExtract) that generate optimized C++ code for common Mumps functions.
4. Temporary Variable Management:
  - \* It uses a pool of temporary string buffers (tmp0, tmp1, etc.) to store intermediate results of the expression evaluation.

- \* `iiitmp` keeps track of the number of temporary buffers used in the current statement, and `memcheck` ensures the buffer limit isn't exceeded.

## 5. Numeric and String Literals:

- \* Numeric literals are canonicalized using `cannon` and then treated as string literals in the generated C++.
- \* String literals are escaped for C++ (e.g., handling `\` and `"`).

## 6. Extract2 Utility: This function (likely in another file) is heavily used to finalise the generated C++ code for an operand, stripping markers like `#S#` (symbol) or `#G#` (global).

### Summary of the Component:

`compiler-parse.c` is the heart of the translation logic. It transforms Mumps' dynamic, string-oriented expressions into static C++ code that maintains the same semantics. The use of temporary buffers and `setjmp/longjmp` (from the main loop) allows the generated code to handle Mumps' flexible control flow and data types within a standard C++ execution environment.

Next, I'll examine some of the command-specific handlers. I'll start with `compiler-set.c` as it's the most common command and involves complex LHS (Left Hand Side) logic.

Next, I'll examine some of the command-specific handlers. I'll start with `compiler-set.c` as it's the most common command and involves complex LHS (Left Hand Side) logic.

- ✓ ReadFile `compiler-set.c`
- ✓ ReadFile `compiler-if.c`
- ✓ ReadFile `compiler-set.c`

- + I've analyzed `compiler-set.c` and `compiler-if.c`. I'll also read `compiler-for.c` to complete the set.

- ✓ ReadFile `compiler-for.c`

- + I've analyzed the core command handlers: `compiler-set.c`, `compiler-if.c`, and `compiler-for.c`.

### Key findings:

#### 1. SET Command (`compiler-set.c`):

- \* Handles assignments to both local variables and global arrays.
- \* LHS (Left Hand Side): It can handle multiple assignments in one statement.
- \* Special Cases: It has complex logic for LHS `$PIECE` and `$EXTRACT`. These are translated into calls to the `Interpret` function at runtime, as they require dynamic modification of the target string.
- \* Merge: Implements the Mumps `MERGE` command (often used for deep copying arrays) by calling a runtime `Merge` function.
- \* Indirection: Supports indirection in the command itself (`s @x=y`) by delegating to the interpreter.

#### 2. IF Command (`compiler-if.c`):

- \* Translates Mumps IF into C++ if statements.
- \* Argumentless IF: Checks the current value of `$TEST` (`svPtr->tpx`).
- \* Argumented IF: Evaluates the expression, updates `svPtr->tpx`, and then branches.
- \* Block Support: It manages `IF_flg` to handle the subsequent commands on the same line (which are part of the "if block" in Mumps).

#### 3. FOR Command (`compiler-for.c`):

- \* This is the most complex handler because Mumps FOR loops have unique semantics (start:increment:limit).
- \* Standard Loop: Translates into a C++ for loop. It handles both positive and negative increments by checking the sign at runtime and choosing the appropriate comparison ( $\geq$  or  $\leq$ ).
- \* List Loops: Supports Mumps FOR i=1,2,3,4:1:10, using setjmp and longjmp to "yield" control back to the loop handler after each iteration of the block.
- \* Argumentless FOR: Translates to a while(1) loop.
- \* Function-based FOR: Optimized for \$ORDER() and \$NEXT() calls, allowing efficient iteration through sparse arrays.
- \* Block Entry: It sets up a do { ... } while(0) block for the commands following the FOR on the same line.

#### Summary of the Translation Logic:

The compiler relies on a combination of standard C++ constructs (for, if) and a custom "continuation" mechanism based on setjmp/longjmp. This allows it to support Mumps' line-oriented execution model and its unique block-scoping rules within a compiled executable. The heavy use of the MSV structure and the Interpret function for "hard" cases ensures compatibility with the interpreter's behavior.

I've now covered the most significant parts of the system:

- Runtime Library: Memory, symbols, globals, math, string tools.
  - Interpreter: Execution loop, parsing, built-in and custom functions.
  - Compiler: Main loop, expression translation, core command handlers.
- ✓ ReadFile compiler-globalarray.c
  - ✓ ReadFile compiler-localarray.c
- ✦ I've analyzed compiler-globalarray.c and compiler-localarray.c, which handle the translation of Mumps array references into C++ code.

#### Key findings:

1. Coded Delimiters: Mumps array references (e.g., ^A(1,"sub")) are translated into a flattened string format using "coded" characters:
  - \* 206 (hex 0xCE): Open parenthesis.
  - \* 207 (hex 0xCF): Close parenthesis.
  - \* 208 (hex 0xD0): Comma.
  - \* This matches the internal format used by the sym\_ and Mglobal functions in the runtime library.
2. GlobalArray (compiler-globalarray.c):
  - \* Handles references starting with ^.
  - \* It generates code to build a temporary string (tmpX) by concatenating the global name and each subscript.
  - \* It uses strxcat and strcat to join parts, correctly handling whether a subscript is a literal or the result of an expression (indicated by #G# or #S# markers).
  - \* The resulting tmpX contains the canonical key used for database operations.
3. LocalArray (compiler-localarray.c):

- \* Handles references that are not globals or special variables, and contain array subscripts (identified by characters > 127, which are the coded delimiters inserted by the parser).
- \* It generates a C++ block `{ char tmp[STR_MAX]; ... }` to reconstruct the full local array reference string.
- \* The final reference is stored in a special `atmp` variable, which is then used by the calling command handler (like `SET` or `WRITE`).

#### Overall System Architecture Summary:

The Mumps Language Processors project is a comprehensive suite for executing Mumps programs, consisting of three main layers:

##### 1. Runtime Library (`libmpscpp`):

- \* Written in C++.
- \* Provides the core data structures and services: multi-level symbol table, SQLite-backed global arrays, string manipulation (`$PIECE`, `$EXTRACT`), and a massive library of "Z" functions for GTK GUIs, bioinformatics, and scientific computing.

##### 2. Interpreter (`mumps`):

- \* Written in C++.
- \* Implements a classic execution loop and recursive-descent expression parser.
- \* Ideal for interactive use and dynamic code execution (`XECUTE`, `indirection`).

##### 3. Compiler (`mumpsc`):

- \* Written in C.
- \* Translates Mumps source code into C++ source code.
- \* Uses `setjmp/longjmp` for flow control and delegates "hard" dynamic features to the interpreter.
- \* The generated C++ is compiled and linked against the Runtime Library for high performance.

This hybrid approach allows the project to provide both the flexibility of an interpreter and the performance of a compiled language, all while sharing a single, powerful runtime backend. The integration with SQLite and GTK makes it a modern and versatile platform for specialized application development.

> final summary

- + This project is a high-performance, modern implementation of the Mumps (M) programming language, comprising a dual-path execution environment: a C++ Interpreter for dynamic execution and a C Compiler that translates Mumps source into optimized C++ code.

##### 1. Architectural Overview

The system is built on a unified C++ Runtime Library (`libmpscpp`) that provides the foundational data structures and services used by both the interpreter and the compiled executables.

- \* Interpreter (`interp-*.cpp`): Implements a classic Mumps execution loop. It is designed for interactive use, debugging, and handling highly dynamic language features like `XECUTE` and `atom-level indirection (@)`.
- \* Compiler (`compiler-*.c`): A single-pass translator that converts Mumps source into C++ source. It leverages `setjmp/longjmp` to emulate Mumps' unique non-local flow control (e.g., `QUIT` from nested blocks) and delegates complex dynamic tasks back to the interpreter's parser at runtime.

- \* **Persistent Storage:** Unlike traditional Mumps systems that use proprietary B-Tree implementations, this project maps the hierarchical Mumps global array model (^GLOBAL) onto a SQLite backend. It flattens the tree into a multi-column relational table, achieving high performance and SQL interoperability.

## 2. Core Language Features

The implementation supports the full standard Mumps language while significantly extending it for modern use cases:

- \* **Symbol Table:** A robust, multi-level system that handles Mumps' unique scoping rules (NEW, KILL), call-by-reference parameters, and nested subroutine execution.
- \* **String Processing:** Highly optimized implementations of \$PIECE and \$EXTRACT for both retrieval and assignment (LHS), which are the workhorses of Mumps data manipulation.
- \* **Numeric Engine:** Support for standard floating-point, extended precision (long double), and arbitrary-precision arithmetic via GMP/MPFR integration.
- \* **Control Flow:** Comprehensive support for Mumps' line-oriented structure, including "Doty" blocks (indented lines starting with .), post-conditionals, and complex FOR loop semantics.

## 3. Modern Extensions ("Z" Functions)

The project transforms Mumps from a legacy database language into a versatile tool for specialized domains:

- \* **GUI Development:** Extensive GTK+ bindings allow developers to build modern Linux graphical interfaces (Dialogs, TreeViews, FileChoosers) directly from Mumps code.
- \* **Bioinformatics:** Native implementations of the Smith-Waterman algorithm for sequence alignment and Boyer-Moore-Gosper for fast string searching.
- \* **Information Retrieval:** Built-in functions for text normalization, stemming, and similarity metrics (Jaccard, Cosine, Soundex, Dice).
- \* **Web & System:** Integrated support for CGI (Common Gateway Interface), HTML generation, and direct shell command execution.

## 4. Implementation Strategy

- \* **Interpreter:** Uses a recursive-descent parser and a large command-switch loop. It maintains state in a MSV (Mumps State Vector) structure, which is passed between all runtime functions.
- \* **Compiler:** Generates C++ code that uses a pool of temporary string buffers (tmp0, tmp1, etc.) for expression evaluation. It minimizes overhead by directly calling C++ runtime functions for arithmetic and symbol lookups while maintaining the "string-is-everything" Mumps philosophy.
- \* **Data Integrity:** Employs canonical key formatting (keyfix) to ensure that array subscripts are handled consistently across local and global storage, regardless of the database backend.

## Summary

The Mumps-Interpreter-Compiler-Library project is a sophisticated bridge between legacy database programming and modern scientific computing. By combining a classic Mumps core with a relational SQLite backend and advanced C++ extensions, it provides a high-performance platform suitable for everything from rapid prototyping and GUI development to complex bioinformatics analysis.