**Ghulam Ishaq Khan Institute of Engineering Sciences and Technology**



**Faculty of Computer Sciences and Engineering**

**Compiler Design Project**

***Submitted by :***

**Khizar Ali Shah 2020196**

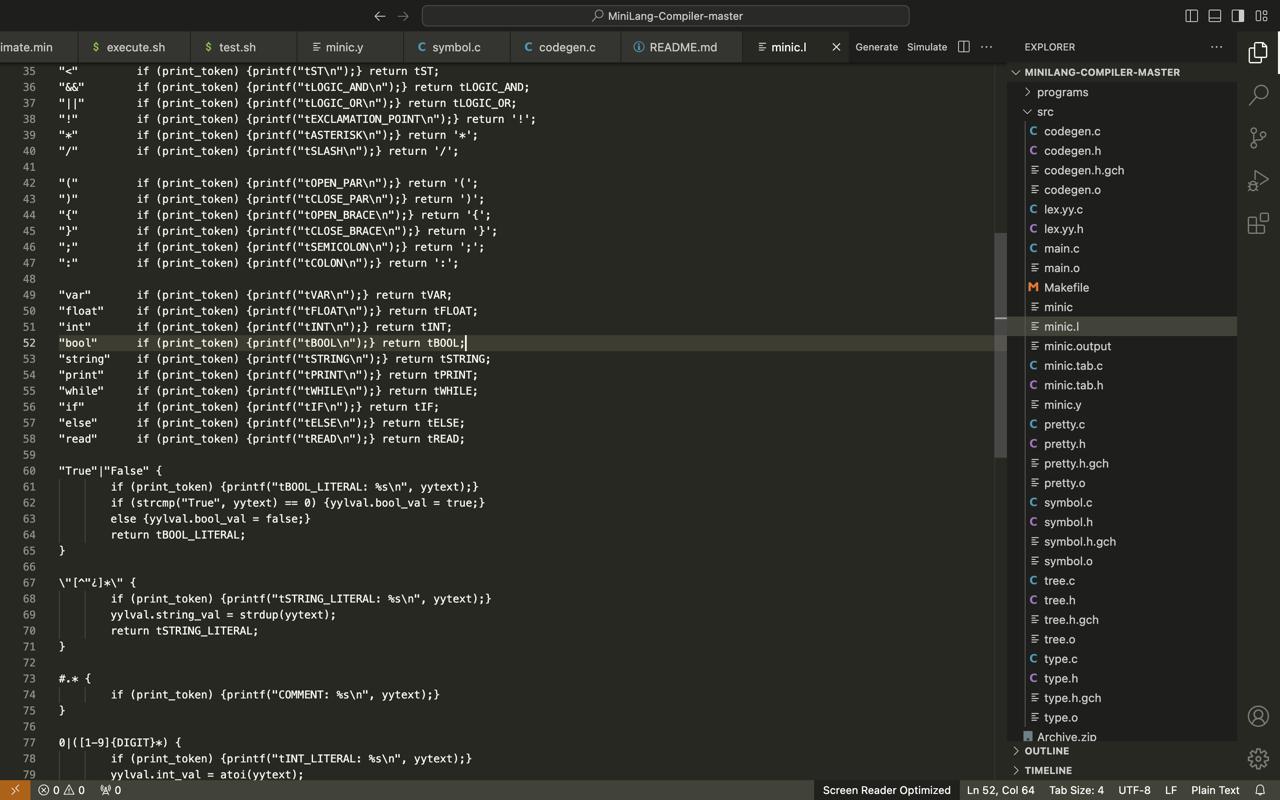
**Afeef Ejaz 2020040**

**Ammar Bin Farrukh 2020468**

***Instructor:***

***Usama Janjua***

**Lexical Analysis (Scanner)**



A lexer, also known as a scanner, is the first stage in processing code written in a programming language. Its job is to break down the input text into smaller meaningful units called tokens.

Here's a breakdown of the code:

**1. Single-character tokens:**

The code defines several functions that handle single-character tokens like "<", "&&", etc.

Each function checks a specific character and if a match is found, it performs two actions:

If print\_token is true (likely for debugging purposes), it prints a descriptive message about the token found.

It returns a specific value representing the token type (e.g., tST for <).

**2. Multi-character tokens (operators):**

The code doesn't use separate functions for multi-character operators like != or <<. It likely relies on the way the lexer reads characters one by one.

By checking for specific character sequences, it can identify these operators and return appropriate token types.

**3. Delimiters:**

Similar to single-character tokens, the code defines functions for delimiters like parentheses (), curly braces {}, etc.

Each function follows the same pattern as single-character tokens - printing a message (if enabled) and returning the token type.

**4. Keywords:**

The code handles keywords like var, int, while, etc. in a similar way.

It compares the current string with the keyword and if a match is found, it prints a message (if enabled) and returns a specific token type representing the keyword (e.g., tVAR for var).

**5. Boolean literals:**

The code handles boolean literals (True and False) differently.

It checks if the current string matches either "True" or "False".

Based on the match, it prints a message (if enabled), sets the boolean value in yylval (likely a global variable used to store the parsed value), and returns the token type tBOOL\_LITERAL.

**6. String literals:**

The code uses a regular expression \"[^"¿]\*\" to match strings.

This regex matches any sequence of characters enclosed in double quotes, except for double quotes themselves and the escape character \.

If a match is found, it prints a message (if enabled), allocates memory for the string using strdup and stores it in yylval, and returns the token type tSTRING\_LITERAL.

**Syntax Analysis (Parser)**

A screenshot of a computer

Description automatically generated

Parsers take the tokens produced by a lexer and build a structured representation of the code, often in the form of a parse tree. Here's a breakdown of the grammar rules:

**1. Program Structure:**

The grammar starts with a rule for program. It defines a program as a single stmt\_list (list of statements). The result ($$) of the rule is assigned the value of the first element ($1).

**2. Statement List:**

The stmt\_list rule can be empty ({$$ = NULL;}), meaning a program can have no statements.

Alternatively, it can be a single statement followed by more statements (stmt stmt\_list). In this case, the result ($$) is created using a function makeSTMT\_LIST that likely combines the first statement ($1) with the following statements ($2).

**3. Statements:**

The grammar defines various types of statements the parser can recognize. Each statement rule has a pattern followed by an action that defines how to build the parse tree.

For example, the rule for a read statement (tREAD '(' tIDENTIFIER ')' ';') matches the tokens tREAD, (, an identifier (tIDENTIFIER), ), and a semicolon ;. The action ({$$ = makeSTMT\_read($3);}) uses a function makeSTMT\_read to create a parse tree node representing a read statement likely with the identifier ($3) as its argument.

**4. if Statements:**

The grammar defines different variations of if statements using three rules for if\_stmt.

The first rule handles a basic if statement with a single block of code.

The second rule handles if-else statements with two code blocks.

The third rule handles if-else if statements with potentially nested if conditions.

Similar to other statement rules, each rule uses appropriate functions (e.g., makeIFSTMT\_ifElseIf) to construct the parse tree nodes representing the specific if statement structure.

**5. Expressions:**

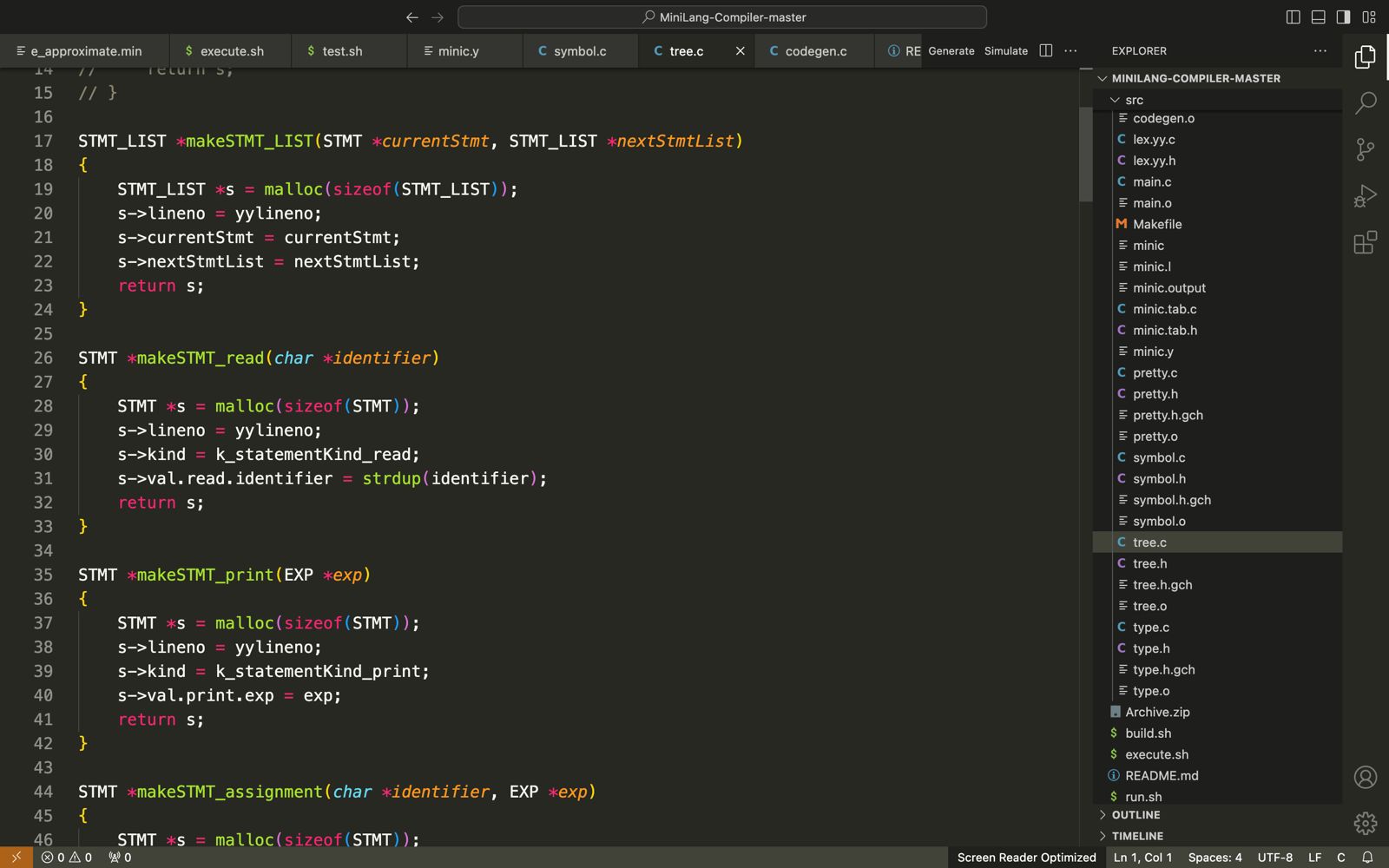
The grammar defines various expressions the parser can recognize.

It includes binary expressions for arithmetic operations (+, -, \*, /) and logical comparisons (==, !=, <, <=).

The grammar also handles unary minus (-), logical NOT (!), and expressions within parentheses.

Each expression rule uses functions like makeEXP\_binary or makeEXP\_unary to create parse tree nodes representing the expression type and its operands.

**Semantic Analysis (AST)**



Building abstract syntax tree (AST) nodes for specific statement types in a compiler. Here's a breakdown of each function:

**1. makeSTMT\_LIST:**

This function creates an AST node for a statement list.

It allocates memory for a new STMT\_LIST structure using malloc.

It sets the line number (lineno) of the statement list to the current line number obtained from the lexer (likely stored in yylineno).

It sets the currentStmt field to the current statement (currentStmt).

It sets the nextStmtList field to the next statement list (nextStmtList). This allows chaining multiple statements together.

Finally, it returns the newly created STMT\_LIST node.

**2. makeSTMT\_read:**

This function creates an AST node for a read statement.

It allocates memory for a new STMT structure using malloc.

It sets the line number (lineno) of the statement to the current line number obtained from the lexer.

It sets the statement kind (kind) to k\_statementKind\_read, indicating it's a read statement.

It allocates memory for the identifier string (identifier) using strdup and stores it in the val.read.identifier field.

strdup is important because the original identifier string might be temporary and the AST node needs its own copy.

Finally, it returns the newly created STMT node representing the read statement.

**3. makeSTMT\_print:**

This function creates an AST node for a print statement.

It allocates memory for a new STMT structure using malloc.

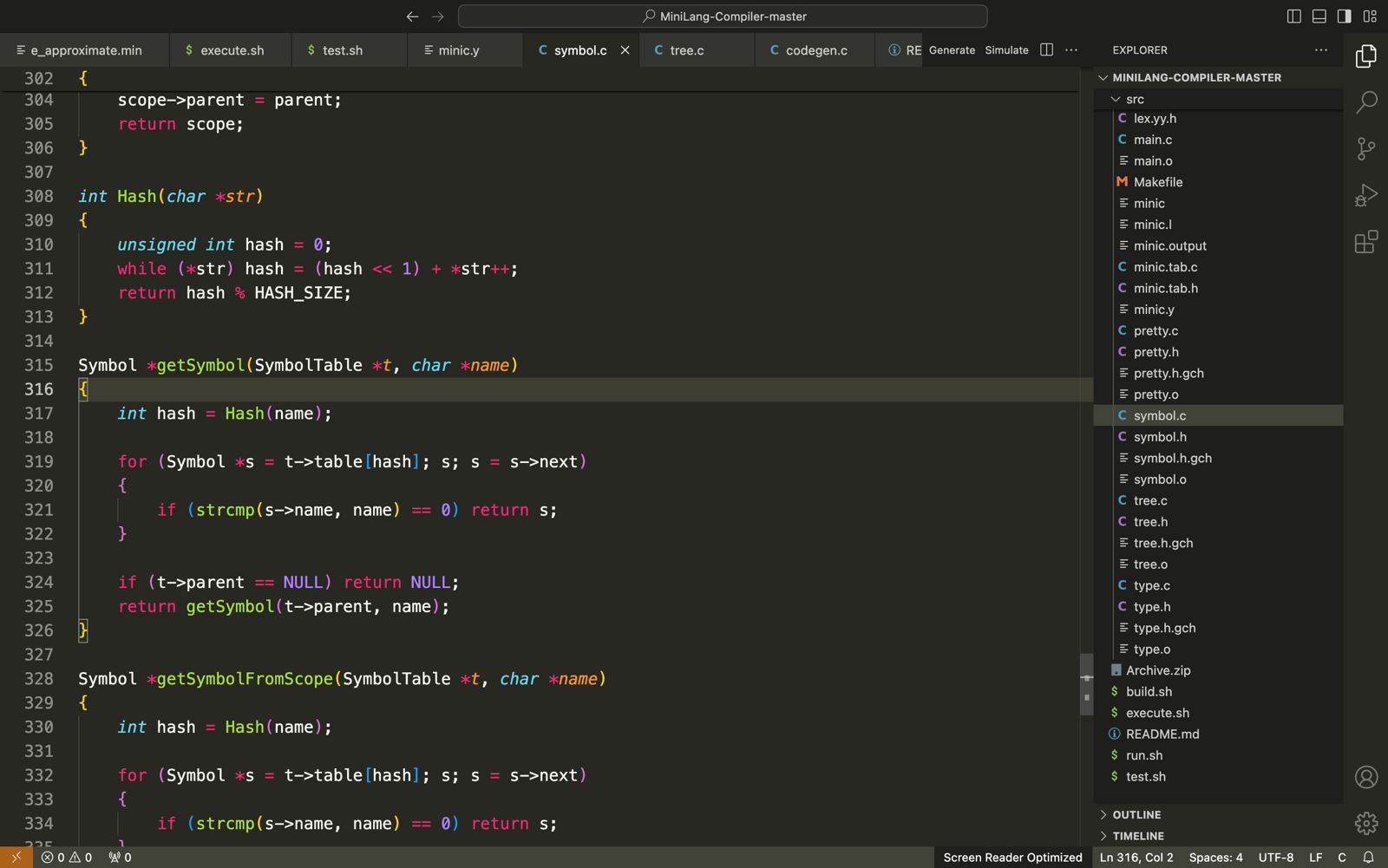
It sets the line number (lineno) of the statement to the current line number obtained from the lexer.

It sets the statement kind (kind) to k\_statementKind\_print, indicating it's a print statement.

It sets the val.print.exp field to the expression (exp) that needs to be printed. The expression tree is likely built by other functions responsible for parsing expressions.

Finally, it returns the newly created STMT node representing the print statement.

**Symbol Table**



A symbol table data structure and related functions for managing symbol information in a compiler. Here's a breakdown of each part:

**1. Hash Function (Hash):**

This function takes a character string (str) as input and calculates a hash value. Hash functions are used to distribute symbols efficiently within a symbol table.

It initializes a variable hash to 0.

It iterates through each character (\*str) in the string.

For each character, it shifts the current hash value one bit to the left (hash << 1).

It adds the current character's ASCII value to the shifted hash value.

Finally, it returns the hash value modulo a constant HASH\_SIZE. This ensures the hash value falls within the range of the symbol table array.

**2. getSymbol (SymbolTable, char):\*\***

This function retrieves a symbol from a symbol table (t) given its name (name).

It calculates the hash value of the symbol name using the Hash function.

It iterates through the linked list of symbols at that hash index (t->table[hash]).

For each symbol (s) in the list, it compares the symbol's name (s->name) with the provided name (name) using strcmp.

If a match is found, it returns the symbol (s).

If no match is found in the current table, the function checks if the symbol table has a parent (t->parent). If it does, it recursively calls getSymbol on the parent table to search for the symbol in a broader scope.

**3. getSymbolFromScope (SymbolTable, char):\*\***

This function is similar to getSymbol but only searches within the current symbol table (t) and doesn't traverse to parent scopes.

It calculates the hash value of the symbol name using the Hash function.

It iterates through the linked list of symbols at that hash index (t->table[hash]).

For each symbol (s) in the list, it compares the symbol's name (s->name) with the provided name (name) using strcmp.

If a match is found, it returns the symbol (s).

If no match is found, it returns NULL.

**4. putSymbol (SymbolTable, char, Type, int):\*\***

This function inserts a new symbol into a symbol table (t).

It calculates the hash value of the symbol name using the Hash function.

It iterates through the linked list of symbols at that hash index.

If a symbol with the same name (strcmp(s->name, name) == 0) is found, it throws an error using a function throwErrorRedeclaredId (likely not shown here) indicating a redeclaration error.

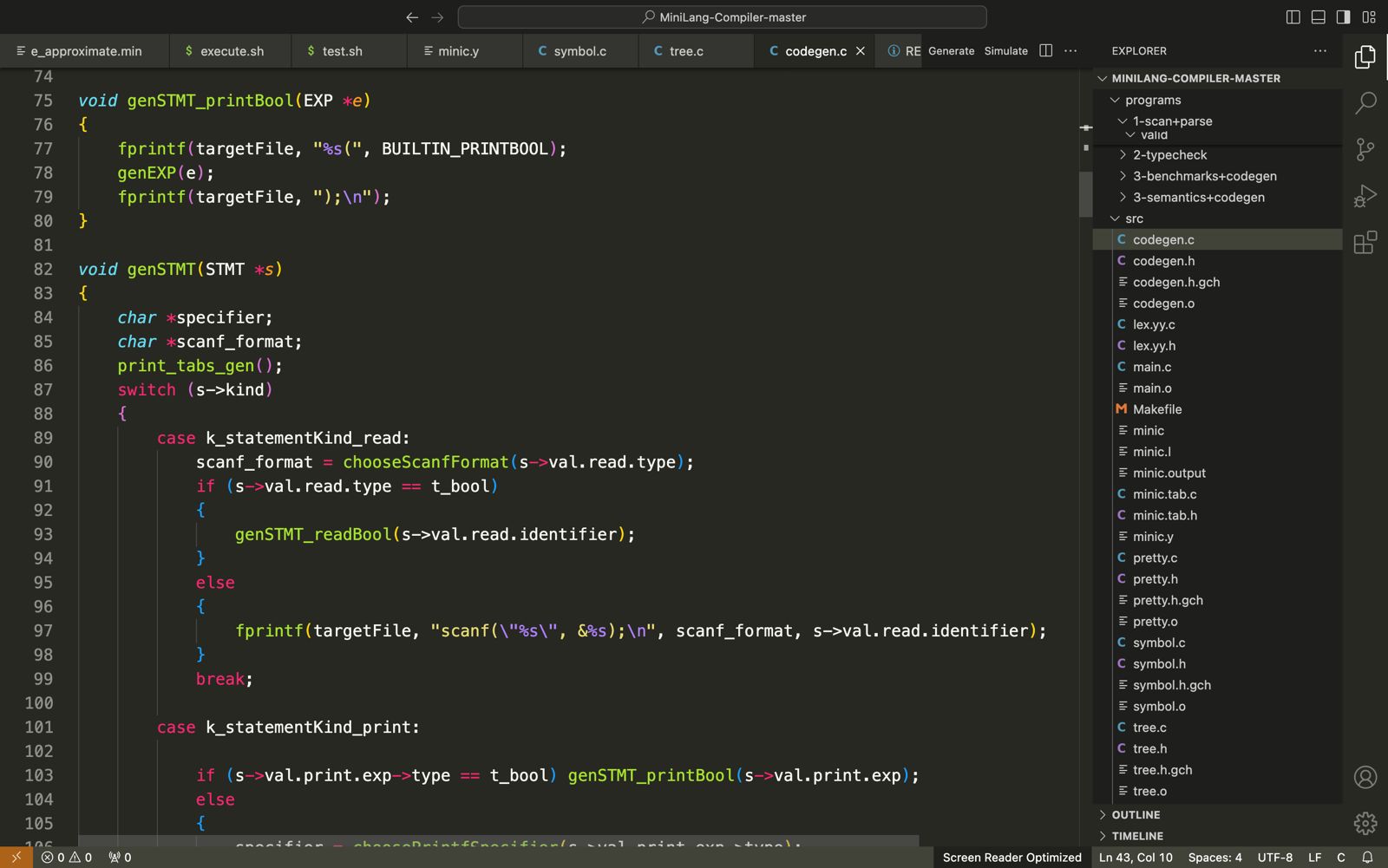
It allocates memory for a new Symbol structure.

It sets the name, type, and next pointer of the new symbol.

It inserts the new symbol at the head of the linked list at the calculated hash index.

Finally, it returns the newly inserted symbol.

**Intermediate Code Generation**



A code generator for a mini-language compiler. It takes an abstract syntax tree (AST) representing the program and generates C code. Let's break down the functionalities of each part:

**1. genProgram:**

This function generates the entire C code for the program.

It takes the AST root (root) and the name of the mini-language source file (name) as input.

It constructs the output filename by replacing the ".min" suffix with ".c" (assuming the mini-language uses ".min" for source files).

It opens the output file (targetFile) for writing.

It checks if the AST root (root) is not null. If it is, it means there's no program to generate.

If root is not null, it further checks if the current statement (root->currentStmt) exists. This handles potential empty programs.

If there's a valid program (root and root->currentStmt are not null), it generates necessary header inclusions for C code (#include statements).

It calls functions to generate code for built-in functions like concat, readBool, and printBool (likely defined elsewhere). These functions are probably placeholders for actual code generation based on the mini-language's built-in functionality.

It starts the main function definition in C.

It increases an indentation counter (num\_tabs\_gen) to control proper formatting of the generated C code (likely not shown here).

It calls genSTMT to generate code for the current statement (root->currentStmt) in the AST.

It recursively calls genSTMT\_LIST to generate code for any subsequent statements in the program represented by the nextStmtList field of the current statement list node.

It prints the closing curly brace and return 0; statement for the main function.

Finally, it closes the output file.

**2. genSTMT\_LIST:**

This function recursively generates code for a list of statements.

If the statement list (s) is not null, it calls genSTMT to generate code for the current statement (s->currentStmt) and then recursively calls itself (genSTMT\_LIST) on the remaining statement list (s->nextStmtList).

**3. genSTMT\_readBool and genSTMT\_printBool:**

These functions likely handle generating code for built-in functions readBool and printBool (specific to the mini-language).

genSTMT\_readBool takes the identifier (id) as input and generates code to call the BUILTIN\_READBOOL function (presumably defined elsewhere) with the identifier as an argument.

genSTMT\_printBool takes the expression (e) as input and generates code to call the BUILTIN\_PRINTBOOL function with the expression tree translated into C code using genEXP (presumably defined elsewhere).

**4. genSTMT:**

This function generates code for a single statement based on its kind (s->kind).

For k\_statementKind\_read:

It determines the appropriate scanf format string based on the variable type (s->val.read.type) using a function chooseScanfFormat (likely defined elsewhere).

If the type is boolean (t\_bool), it calls genSTMT\_readBool for built-in boolean reading.

Otherwise, it generates a scanf statement to read input based on the type and identifier.

For k\_statementKind\_print:

If the expression type (s->val.print.exp->type) is boolean, it calls genSTMT\_printBool to handle built-in boolean printing.

Otherwise, it determines the appropriate printf format specifier using a function choosePrintfSpecifier (likely defined elsewhere).

It generates a printf statement with the format specifier and calls genEXP to translate the expression tree into C code.

For k\_statementKind\_assignment:

It generates an assignment statement in C using the identifier (s->val.assignment.identifier) and the expression (s->val.assignment.exp) translated into C code using genEXP.

**Execution**

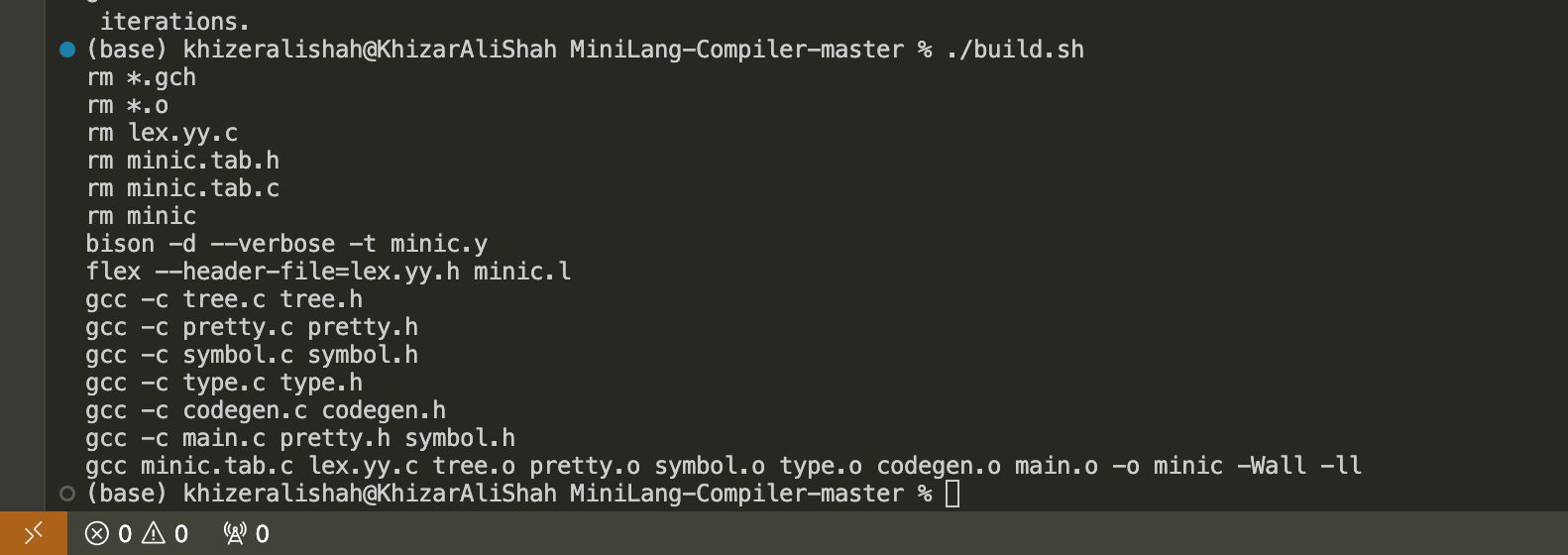
A screenshot of a computer program

Description automatically generated

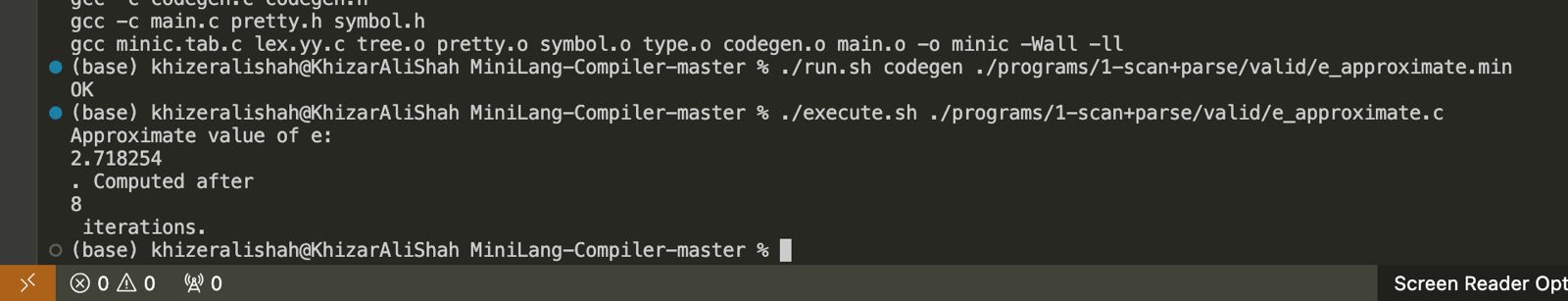
approximates the value of the mathematical constant e by summing the series

1/n! until the change between iterations is less than a specified threshold (0.001 in this case). The program initializes variables for the sum, factorial, and iteration count. It then enters a loop, updating the sum and factorial in each iteration, and checks if the change between the current and previous sum is below the threshold to determine when to stop. The resulting approximate value of e and the number of iterations required are then printed.

**Build**



**Run & Execute**



Completing this compiler project as a team has been a remarkable journey of collaboration, innovation, and persistence. We have not only built a tool that embodies our collective expertise and dedication but also forged stronger bonds and enhanced our skills in the process. This achievement stands as a testament to our shared vision and unwavering commitment to excellence. Here's to the countless hours of coding, debugging, and problem-solving—together, we have created something truly exceptional.