```c
// parser.c
// Recursive Descent Parser for Control Structures and Nested Expressions
// Course: Programming Languages and Structures
// Code by: Md. Alamin
// Student ID: 21303134

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <setjmp.h>  // jmp_buf, setjmp, and longjmp
#include <unistd.h>  // dup, dup2, and close functions

// --- Configuration ---

#define MAX_SYMBOLS 100
int g_student_ltd_value = 134; // Default value for LTD (Last Three Digits of Student ID)
// --- Global Variables for Lexer and Parser ---
static const char *g_source_code; // Start of the source code
static const char *g_source_ptr;  // Pointer to the current character

// =============1. Lexer (Scanner) Implementation============== start
// --- Token Definitions ---
typedef enum {
    TOKEN_EOF,
    TOKEN_ERROR,

    // Keywords
    TOKEN_IF,
    TOKEN_ELSE,
    TOKEN_WHILE,

    // Symbols
    TOKEN_LBRACE,     // {
    TOKEN_RBRACE,     // }
    TOKEN_LPAREN,     // (
    TOKEN_RPAREN,     // )
    TOKEN_SEMICOLON,  // ;

    // Operators
    TOKEN_PLUS,       // +
    TOKEN_MINUS,      // -
    TOKEN_MULTIPLY,   // *
    TOKEN_DIVIDE,     // /

    // Relational Operators
    TOKEN_EQ,         // ==
    TOKEN_NEQ,        // !=
    TOKEN_LT,         // <
    TOKEN_GT,         // >
    TOKEN_LTE,        // <=
    TOKEN_GTE,        // >=

    // Literals and Identifiers
    TOKEN_NUMBER,
    TOKEN_IDENTIFIER,
    TOKEN_LTD         // Special identifier "LTD"
} TokenType;

const char* token_type_to_string(TokenType type) {
    switch (type) {
        case TOKEN_EOF: return "EOF";
        case TOKEN_ERROR: return "ERROR";
        case TOKEN_IF: return "IF";
        case TOKEN_ELSE: return "ELSE";
        case TOKEN_WHILE: return "WHILE";
        case TOKEN_LBRACE: return "LBRACE ('{')";
        case TOKEN_RBRACE: return "RBRACE ('}')";
        case TOKEN_LPAREN: return "LPAREN ('(')";
        case TOKEN_RPAREN: return "RPAREN (')')";
        case TOKEN_SEMICOLON: return "SEMICOLON (';')";
        case TOKEN_PLUS: return "PLUS ('+')";
        case TOKEN_MINUS: return "MINUS ('-')";
        case TOKEN_MULTIPLY: return "MULTIPLY ('*')";
        case TOKEN_DIVIDE: return "DIVIDE ('/')";
        case TOKEN_EQ: return "EQ ('==')";
        case TOKEN_NEQ: return "NEQ ('!=')";
        case TOKEN_LT: return "LT ('<')";
        case TOKEN_GT: return "GT ('>')";
        case TOKEN_LTE: return "LTE ('<=')";
        case TOKEN_GTE: return "GTE ('>=')";
        case TOKEN_NUMBER: return "NUMBER";
        case TOKEN_IDENTIFIER: return "IDENTIFIER";
        case TOKEN_LTD: return "LTD";
```

```
85          default: return "UNKNOWN_TOKEN";
86      }
87  }
88
89  typedef struct {
90      TokenType type;
91      char value[100]; // Buffer for number value or identifier name
92      int line;        // Line number where the token starts
93      int col;         // Column number where the token starts
94  } Token;
95
96  typedef struct {
97      char name[100];
98      int value;
99  } Symbol;
100
101 Symbol g_symbol_table[MAX_SYMBOLS];
102 int g_symbol_count = 0;
103
104 // --- Global Variables for Lexer and Parser ---
105 static Token g_current_token;      // The current token being processed by the parser
106 static int g_current_line = 1;     // Current line number in the source
107 static int g_current_col = 1;      // Current column number in the source
108 static int g_start_col_for_token = 1; // Column where the current token began
109
110 // --- Forward Declarations for Parser Functions ---
111 static void program();
112 static void block();
113 static void statement();
114 static void if_statement();
115 static void while_statement();
116 static void condition();
117 static void relational_operator();
118 static void expression();
119 static void term();
120 static void factor();
121
122 // Forward declarations for evaluator functions
123 static int eval_expression();
124 static int eval_term();
125 static int eval_factor();
126 // Error handling forward declaration
127 static void error_at_current_token(const char* message);
128
129 // Pre-defined test cases
130 const char* test_cases[] = {
131     // Valid test cases
132     "{ if (a == LTD) { while (b < 100) { (a + b) * (b - LTD); } } else { (x + y) * (a -
    b); } }",
133     "{ a + b; }",
134     "{ if (x > 5) { y + 10; } }",
135     "{ while (i <= 10) { sum + i; i + 1; } }",
136
137     // Invalid test cases
138     "{ a + b }", // Missing semicolon
139     "{ if (a == b) { a + b; }", // Mismatched brackets
140     "{ 3a + 5; }", // Invalid identifier
141     "{ if (a > b) if (c < d) { x; } }", // Nested if without braces for outer if
142     "{ else { x; } }" // else without if
143 };
144
145 // read input from console
146 char* read_from_console() {
147     printf("Enter program (end with Ctrl+D on Unix/Linux or Ctrl+Z on Windows):\n");
148
149     // Initial buffer size
150     size_t bufsize = 1024;
151     char* buffer = (char*)malloc(bufsize);
152     if (!buffer) {
153         fprintf(stderr, "Memory allocation failed\n");
154         return NULL;
155     }
156
157     size_t position = 0;
158     int c;
159
160     while ((c = getchar()) != EOF) {
161         // Resize buffer if needed
162         if (position >= bufsize - 1) {
163             bufsize *= 2;
164             char* new_buffer = (char*)realloc(buffer, bufsize);
165             if (!new_buffer) {
166                 fprintf(stderr, "Memory reallocation failed\n");
167                 free(buffer);
```

```
                return NULL;
            }
            buffer = new_buffer;
        }

        buffer[position++] = (char)c;
    }

    // Null-terminate the string
    buffer[position] = '\0';
    return buffer;
}

// Function to look up or add a symbol
static int get_symbol_value(const char* name) {
    // Look for existing symbol
    for (int i = 0; i < g_symbol_count; i++) {
        if (strcmp(g_symbol_table[i].name, name) == 0) {
            return g_symbol_table[i].value;
        }
    }

    // Add new symbol with dummy value (0)
    if (g_symbol_count < MAX_SYMBOLS) {
        strcpy(g_symbol_table[g_symbol_count].name, name);
        g_symbol_table[g_symbol_count].value = 0; // Default value
        return g_symbol_table[g_symbol_count++].value;
    } else {
        error_at_current_token("Symbol table overflow");
        return 0;
    }
}


// =============3. Error Handling============== start

// --- Error Handling ---
static void error_at_current_token(const char* message) {
    fprintf(stderr, "Syntax Error on line %d, col %d: %s\n", g_current_token.line, g_current_token.col, message);
    fprintf(stderr, "Near token: '%s' (Type: %s)\n", g_current_token.value, token_type_to_string(g_current_token.type));

    // Find the beginning of the line
    const char* line_start = g_source_ptr;
    while (line_start > g_source_code && *(line_start-1) != '\n') {
        line_start--;
    }

    // Find the end of the line
    const char* line_end = g_source_ptr;
    while (*line_end != '\0' && *line_end != '\n') {
        line_end++;
    }

    // Print the line
    fprintf(stderr, "Line %d: ", g_current_token.line);
    fprintf(stderr, "%.*s\n", (int)(line_end - line_start), line_start);

    // Print a caret pointing to the error position
    fprintf(stderr, "%*s^\n", g_current_token.col - 1, "");

    exit(EXIT_FAILURE);
}

// --- Lexer Implementation ---
static Token make_token(TokenType type, const char* value_str, int line, int col) {
    Token token;
    token.type = type;
    if (value_str != NULL) {
        strncpy(token.value, value_str, sizeof(token.value) - 1);
        token.value[sizeof(token.value) - 1] = '\0';
    } else {
        token.value[0] = '\0';
    }
    token.line = line;
    token.col = col;
    return token;
}

static void skip_whitespace_and_comments() {
    while (*g_source_ptr != '\0') {
        if (isspace((unsigned char)*g_source_ptr)) {
            if (*g_source_ptr == '\n') {
```

```
250                 g_current_line++;
251                 g_current_col = 1;
252             } else {
253                 g_current_col++;
254             }
255             g_source_ptr++;
256         }
257         // Handle C-style comments
258         else if (*g_source_ptr == '/' && *(g_source_ptr + 1) == '*') {
259             g_source_ptr += 2; // Skip /*
260             g_current_col += 2;
261 
262             while (!(*g_source_ptr == '*' && *(g_source_ptr + 1) == '/') && *g_source_ptr
    != '\0') {
263                 if (*g_source_ptr == '\n') {
264                     g_current_line++;
265                     g_current_col = 1;
266                 } else {
267                     g_current_col++;
268                 }
269                 g_source_ptr++;
270             }
271 
272             if (*g_source_ptr == '\0') {
273                 // Unclosed comment
274                 error_at_current_token("Unclosed comment detected");
275             } else {
276                 g_source_ptr += 2; // Skip */
277                 g_current_col += 2;
278             }
279         }
280         // Handle C++-style comments
281         else if (*g_source_ptr == '/' && *(g_source_ptr + 1) == '/') {
282             g_source_ptr += 2; // Skip //
283             g_current_col += 2;
284 
285             while (*g_source_ptr != '\n' && *g_source_ptr != '\0') {
286                 g_source_ptr++;
287                 g_current_col++;
288             }
289         }
290         else {
291             break; // Not whitespace or comment
292         }
293     }
294 }
295 
296 static Token lexer_get_next_token_internal() {
297     skip_whitespace_and_comments();
298     g_start_col_for_token = g_current_col; // Record column at start of token
299 
300     if (*g_source_ptr == '\0') {
301         return make_token(TOKEN_EOF, "EOF", g_current_line, g_start_col_for_token);
302     }
303 
304     char current_char = *g_source_ptr;
305     char next_char = *(g_source_ptr + 1);
306 
307     // Multi-character operators (==, !=, <=, >=)
308     if (current_char == '=' && next_char == '=') { g_source_ptr += 2; g_current_col += 2;
    return make_token(TOKEN_EQ, "==", g_current_line, g_start_col_for_token); }
309     if (current_char == '!' && next_char == '=') { g_source_ptr += 2; g_current_col += 2;
    return make_token(TOKEN_NEQ, "!=", g_current_line, g_start_col_for_token); }
310     if (current_char == '<' && next_char == '=') { g_source_ptr += 2; g_current_col += 2;
    return make_token(TOKEN_LTE, "<=", g_current_line, g_start_col_for_token); }
311     if (current_char == '>' && next_char == '=') { g_source_ptr += 2; g_current_col += 2;
    return make_token(TOKEN_GTE, ">=", g_current_line, g_start_col_for_token); }
312 
313     // Single-character symbols and operators
314     char single_char_val[2] = {current_char, '\0'};
315     g_source_ptr++; g_current_col++;
316     switch (current_char) {
317         case '{': return make_token(TOKEN_LBRACE, single_char_val, g_current_line,
    g_start_col_for_token);
318         case '}': return make_token(TOKEN_RBRACE, single_char_val, g_current_line,
    g_start_col_for_token);
319         case '(': return make_token(TOKEN_LPAREN, single_char_val, g_current_line,
    g_start_col_for_token);
320         case ')': return make_token(TOKEN_RPAREN, single_char_val, g_current_line,
    g_start_col_for_token);
321         case ';': return make_token(TOKEN_SEMICOLON, single_char_val, g_current_line,
    g_start_col_for_token);
322         case '+': return make_token(TOKEN_PLUS, single_char_val, g_current_line,
    g_start_col_for_token);
```

```
323             case '-': return make_token(TOKEN_MINUS, single_char_val, g_current_line,
    g_start_col_for_token);
324             case '*': return make_token(TOKEN_MULTIPLY, single_char_val, g_current_line,
    g_start_col_for_token);
325             case '/': return make_token(TOKEN_DIVIDE, single_char_val, g_current_line,
    g_start_col_for_token);
326             case '<': return make_token(TOKEN_LT, single_char_val, g_current_line,
    g_start_col_for_token);
327             case '>': return make_token(TOKEN_GT, single_char_val, g_current_line,
    g_start_col_for_token);
328         }
329         // Backtrack if not a recognized single character
330         g_source_ptr--; g_current_col--;
331
332         // Numbers: <digit> { <digit> }
333         if (isdigit((unsigned char)current_char)) {
334             char num_buffer[100];
335             int i = 0;
336             num_buffer[i++] = current_char;
337             g_source_ptr++; g_current_col++;
338             while (*g_source_ptr != '\0' && isdigit((unsigned char)*g_source_ptr)) {
339                 if (i < sizeof(num_buffer) - 1) num_buffer[i++] = *g_source_ptr;
340                 g_source_ptr++; g_current_col++;
341             }
342             num_buffer[i] = '\0';
343             return make_token(TOKEN_NUMBER, num_buffer, g_current_line, g_start_col_for_token);
344         }
345
346         // Identifiers and Keywords: <letter> { <letter> | <digit> } (allow underscore)
347         if (isalpha((unsigned char)current_char) || current_char == '_') {
348             char id_buffer[100];
349             int i = 0;
350             id_buffer[i++] = current_char;
351             g_source_ptr++; g_current_col++;
352             while (*g_source_ptr != '\0' && (isalnum((unsigned char)*g_source_ptr) ||
    *g_source_ptr == '_')) {
353                 if (i < sizeof(id_buffer) - 1) id_buffer[i++] = *g_source_ptr;
354                 g_source_ptr++; g_current_col++;
355             }
356             id_buffer[i] = '\0';
357
358             // Check for keywords
359             if (strcmp(id_buffer, "if") == 0) return make_token(TOKEN_IF, id_buffer,
    g_current_line, g_start_col_for_token);
360             if (strcmp(id_buffer, "else") == 0) return make_token(TOKEN_ELSE, id_buffer,
    g_current_line, g_start_col_for_token);
361             if (strcmp(id_buffer, "while") == 0) return make_token(TOKEN_WHILE, id_buffer,
    g_current_line, g_start_col_for_token);
362             if (strcmp(id_buffer, "LTD") == 0) return make_token(TOKEN_LTD, id_buffer,
    g_current_line, g_start_col_for_token);
363
364             return make_token(TOKEN_IDENTIFIER, id_buffer, g_current_line,
    g_start_col_for_token);
365         }
366
367         // If no rule matches, it's an unrecognized character
368         char error_char_val[2] = {current_char, '\0'};
369         g_source_ptr++; g_current_col++; // Consume the erroneous character to avoid infinite
    loop
370         Token err_token = make_token(TOKEN_ERROR, error_char_val, g_current_line,
    g_start_col_for_token);
371         // Error will be reported by advance()
372         return err_token;
373     }
374
375     // =============1. Lexer (Scanner) Implementation============== end
376
377     // --- Parser Helper Functions ---
378     // Consumes the current token and gets the next one from the lexer.
379     static void advance() {
380         g_current_token = lexer_get_next_token_internal();
381         if (g_current_token.type == TOKEN_ERROR) {
382             char error_msg[150];
383             sprintf(error_msg, "Lexical error: Unrecognized character '%s'",
    g_current_token.value);
384             error_at_current_token(error_msg); // This will exit
385         }
386     }
387
388     // Checks if the current token matches the expected type.
389     // If yes, consumes it (advances). If no, reports an error.
390     static void eat(TokenType expected_type, const char* error_message) {
391         if (g_current_token.type == expected_type) {
392             advance();
```

```
393         } else {
394             char full_error_message[256];
395             sprintf(full_error_message, "%s. Expected %s, but got %s ('%s').",
396                     error_message,
397                     token_type_to_string(expected_type),
398                     token_type_to_string(g_current_token.type),
399                     g_current_token.value);
400             error_at_current_token(full_error_message);
401         }
402     }
403     // =============3. Error Handling============== end
404 
405     // =============2. Recursive Descent Parser============== start
406     // --- Recursive Descent Parser Functions ---
407     // <program> -> <block>
408     static void program() {
409         printf("Parsing <program>...\n");
410         block();
411         if (g_current_token.type != TOKEN_EOF) {
412             error_at_current_token("Expected end of input (EOF) after program block, but found
    more tokens.");
413         }
414         printf("Finished parsing <program>.\n");
415     }
416 
417     // <block> -> "{" { <statement> } "}"
418     static void block() {
419         printf("Parsing <block>...\n");
420         eat(TOKEN_LBRACE, "Expected '{' to start a block");
421         while (g_current_token.type != TOKEN_RBRACE && g_current_token.type != TOKEN_EOF) {
422             // Check if the current token can start a statement
423             TokenType tt = g_current_token.type;
424             if (tt == TOKEN_IF || tt == TOKEN_WHILE || // Keywords for statements
425                 tt == TOKEN_LPAREN ||                  // Start of ( <expression> ) ;
426                 tt == TOKEN_IDENTIFIER || tt == TOKEN_NUMBER || tt == TOKEN_LTD) { // Start
    of <expression> ;
427                 statement();
428             } else {
429                 error_at_current_token("Invalid token inside block. Expected a statement or
    '}'.");
430                 break;
431             }
432         }
433         eat(TOKEN_RBRACE, "Expected '}' to end a block");
434         printf("Finished parsing <block>.\n");
435     }
436 
437     // <statement> -> <if-statement> | <while-statement> | <expression> ";"
438     static void statement() {
439         printf("Parsing <statement> (current token: %s)...\n",
    token_type_to_string(g_current_token.type));
440         if (g_current_token.type == TOKEN_IF) {
441             if_statement();
442         } else if (g_current_token.type == TOKEN_WHILE) {
443             while_statement();
444         } else if (g_current_token.type == TOKEN_LPAREN ||
445                    g_current_token.type == TOKEN_IDENTIFIER ||
446                    g_current_token.type == TOKEN_NUMBER ||
447                    g_current_token.type == TOKEN_LTD) {
448             expression();
449             eat(TOKEN_SEMICOLON, "Expected ';' after expression statement");
450         } else {
451             error_at_current_token("Invalid start of a statement. Expected 'if', 'while', or
    an expression.");
452         }
453         printf("Finished parsing <statement>.\n");
454     }
455 
456     // <if-statement> -> "if" "(" <condition> ")" <block> [ "else" <block> ]
457     static void if_statement() {
458         printf("Parsing <if-statement>...\n");
459         eat(TOKEN_IF, "Expected 'if' keyword");
460         eat(TOKEN_LPAREN, "Expected '(' after 'if'");
461         condition();
462         eat(TOKEN_RPAREN, "Expected ')' after if-condition");
463         block();
464         if (g_current_token.type == TOKEN_ELSE) {
465             eat(TOKEN_ELSE, "Expected 'else' keyword");
466             block();
467         }
468         printf("Finished parsing <if-statement>.\n");
469     }
470 
471     // <while-statement> -> "while" "(" <condition> ")" <block>
```

```c
static void while_statement() {
    printf("Parsing <while-statement>...\n");
    eat(TOKEN_WHILE, "Expected 'while' keyword");
    eat(TOKEN_LPAREN, "Expected '(' after 'while'");
    condition();
    eat(TOKEN_RPAREN, "Expected ')' after while-condition");
    block();
    printf("Finished parsing <while-statement>.\n");
}

// <condition> -> <expression> <relational-operator> <expression>
static void condition() {
    printf("Parsing <condition>...\n");
    expression();
    relational_operator();
    expression();
    printf("Finished parsing <condition>.\n");
}

// <relational-operator> -> "==" | "!=" | "<" | ">" | "<=" | ">="
static void relational_operator() {
    printf("Parsing <relational-operator> (current token: %s)...\n", g_current_token.value);
    switch (g_current_token.type) {
        case TOKEN_EQ:
        case TOKEN_NEQ:
        case TOKEN_LT:
        case TOKEN_GT:
        case TOKEN_LTE:
        case TOKEN_GTE:
            printf("Recognized relational operator: %s\n", g_current_token.value);
            advance(); // Consume the operator
            break;
        default:
            error_at_current_token("Expected a relational operator (e.g., ==, <, >=)");
    }
    printf("Finished parsing <relational-operator>.\n");
}

// <expression> -> <term> { ("+" | "-") <term> }
static void expression() {
    printf("Parsing <expression>...\n");
    term();
    while (g_current_token.type == TOKEN_PLUS || g_current_token.type == TOKEN_MINUS) {
        printf("Recognized operator in expression: %s\n", g_current_token.value);
        advance(); // Consume '+' or '-'
        term();
    }
    printf("Finished parsing <expression>.\n");
}

// <term> -> <factor> { ("*" | "/") <factor> }
static void term() {
    printf("Parsing <term>...\n");
    factor();
    while (g_current_token.type == TOKEN_MULTIPLY || g_current_token.type == TOKEN_DIVIDE) {
        printf("Recognized operator in term: %s\n", g_current_token.value);
        advance(); // Consume '*' or '/'
        factor();
    }
    printf("Finished parsing <term>.\n");
}

// <factor> -> <number> | <identifier> | "LTD" | "(" <expression> ")"
static void factor() {
    printf("Parsing <factor> (current token type: %s, value: '%s')...\n",
token_type_to_string(g_current_token.type), g_current_token.value);
    if (g_current_token.type == TOKEN_NUMBER) {
        printf("Recognized number: %s\n", g_current_token.value);
        eat(TOKEN_NUMBER, "Error processing number in factor."); // eat already advances
    } else if (g_current_token.type == TOKEN_IDENTIFIER) {
        printf("Recognized identifier: %s\n", g_current_token.value);
        eat(TOKEN_IDENTIFIER, "Error processing identifier in factor.");
    } else if (g_current_token.type == TOKEN_LTD) {
        printf("Recognized LTD, substituting with value: %d\n", g_student_ltd_value);
        eat(TOKEN_LTD, "Error processing LTD in factor.");
    } else if (g_current_token.type == TOKEN_LPAREN) {
        eat(TOKEN_LPAREN, "Expected '(' for sub-expression in factor");
        expression();
        eat(TOKEN_RPAREN, "Expected ')' after sub-expression in factor");
    } else {
        char error_msg[200];
        sprintf(error_msg, "Invalid factor. Expected number, identifier, LTD, or '('. Got
token type %s ('%s')",
                token_type_to_string(g_current_token.type), g_current_token.value);
```

```
554             error_at_current_token(error_msg);
555         }
556         printf("Finished parsing <factor>.\n");
557     }
558
559     // =============2. Recursive Descent Parser============== end
560
561
562     // =============4. Expression Evaluation============== start
563
564     // Example implementation for expression evaluation
565     static int eval_term() {
566         int result = eval_factor();
567
568         while (g_current_token.type == TOKEN_MULTIPLY || g_current_token.type == TOKEN_DIVIDE) {
569             TokenType op = g_current_token.type;
570             advance(); // Consume '*' or '/'
571             int factor_value = eval_factor();
572
573             if (op == TOKEN_MULTIPLY) {
574                 result *= factor_value;
575             } else { // DIVIDE
576                 if (factor_value == 0) {
577                     error_at_current_token("Division by zero");
578                 }
579                 result /= factor_value;
580             }
581         }
582
583         return result;
584     }
585
586     static int eval_expression() {
587         int result = eval_term();
588
589         while (g_current_token.type == TOKEN_PLUS || g_current_token.type == TOKEN_MINUS) {
590             TokenType op = g_current_token.type;
591             advance(); // Consume '+' or '-'
592             int term_value = eval_term();
593
594             if (op == TOKEN_PLUS) {
595                 result += term_value;
596             } else { // MINUS
597                 result -= term_value;
598             }
599         }
600
601         return result;
602     }
603
604     static int eval_factor() {
605         int result = 0;
606
607         if (g_current_token.type == TOKEN_NUMBER) {
608             result = atoi(g_current_token.value);
609             eat(TOKEN_NUMBER, "Error processing number in factor evaluation");
610         } else if (g_current_token.type == TOKEN_IDENTIFIER) {
611             result = get_symbol_value(g_current_token.value);
612             eat(TOKEN_IDENTIFIER, "Error processing identifier in factor evaluation");
613         } else if (g_current_token.type == TOKEN_LTD) {
614             result = g_student_ltd_value;
615             eat(TOKEN_LTD, "Error processing LTD in factor evaluation");
616         } else if (g_current_token.type == TOKEN_LPAREN) {
617             eat(TOKEN_LPAREN, "Expected '(' for sub-expression in factor evaluation");
618             result = eval_expression();
619             eat(TOKEN_RPAREN, "Expected ')' after sub-expression in factor evaluation");
620         } else {
621             error_at_current_token("Invalid factor in evaluation");
622         }
623
624         return result;
625     }
626
627     // =============4. Expression Evaluation============== end
628
629     // --- Initialization and Main Driver ---
630     static void initialize_parser(const char* source_code) {
631         g_source_code = source_code;
632         g_source_ptr = source_code;
633         g_current_line = 1;
634         g_current_col = 1;
635         g_start_col_for_token = 1;
636         g_symbol_count = 0;
637
```

```
638         // Load the first token to prime the parser
639         advance();
640     }
641
642     // Function to read entire file into a string
643     char* read_file_to_string(const char* filename) {
644         FILE *file = fopen(filename, "rb"); // Open in binary mode to correctly get length
645         if (!file) {
646             perror("Error opening file");
647             return NULL;
648         }
649
650         fseek(file, 0, SEEK_END);
651         long length = ftell(file);
652         fseek(file, 0, SEEK_SET);
653
654         char *buffer = (char*)malloc(length + 1);
655         if (!buffer) {
656             fprintf(stderr, "Memory allocation failed for file buffer\n");
657             fclose(file);
658             return NULL;
659         }
660
661         if (fread(buffer, 1, length, file) != (size_t)length) {
662             fprintf(stderr, "Error reading file\n");
663             fclose(file);
664             free(buffer);
665             return NULL;
666         }
667         buffer[length] = '\0';
668         fclose(file);
669         return buffer;
670     }
671
672     // =============5. Test Case Suite============== start
673
674     // Process a single test case
675     static void process_test_case(const char* test_input, int test_number, int
    is_valid_expected) {
676         printf("\n\n----------------------------------------\n");
677         printf("TEST CASE %d: %s\n", test_number, is_valid_expected ? "VALID" : "INVALID");
678         printf("----------------------------------------\n");
679         printf("Input: %s\n\n", test_input);
680
681         // Reinitialize parser state
682         initialize_parser(test_input);
683
684         int success = 1;
685
686         // Skip the regular program call if we expect the test to fail
687         if (is_valid_expected) {
688             // Try to parse, but catch errors
689             int old_stderr = dup(STDERR_FILENO);
690             freopen("/dev/null", "w", stderr); // Redirect stderr to prevent error messages
    printing
691
692             // Use a setjmp/longjmp to simulate try/catch
693             jmp_buf env;
694             int error_code = setjmp(env);
695
696             if (!error_code) {
697                 program(); // Start parsing
698                 printf("✓ Program parsed successfully!\n");
699             } else {
700                 success = 0;
701                 printf("✗ Parsing failed unexpectedly!\n");
702             }
703
704             // Restore stderr
705             fflush(stderr);
706             dup2(old_stderr, STDERR_FILENO);
707             close(old_stderr);
708         } else {
709             // For invalid cases, we expect an error
710             int old_stderr = dup(STDERR_FILENO);
711             freopen("/dev/null", "w", stderr); // Redirect stderr
712
713             // Use a setjmp/longjmp to simulate try/catch
714             jmp_buf env;
715             if (setjmp(env) == 0) {
716                 program(); // Start parsing
717                 printf("✗ Expected parsing to fail, but it succeeded!\n");
718                 success = 0;
719             } else {
```

```
            printf("✓ Parsing failed as expected for invalid input.\n");
        }

        // Restore stderr
        fflush(stderr);
        dup2(old_stderr, STDERR_FILENO);
        close(old_stderr);
    }

    printf("\nTest result: %s\n", success ? "PASS" : "FAIL");
}

// =============5. Test Case Suite============== start


// display menu for choosing test method
void display_interactive_menu() {
    printf("\n=== Recursive Descent Parser - Interactive Menu ===\n");
    printf("1. Enter code via console input\n");
    printf("2. Read code from a file\n");
    printf("3. Run test suite\n");
    printf("4. Use default test case\n");
    printf("5. Change LTD value (currently: %d)\n", g_student_ltd_value);
    printf("6. Exit\n");
    printf("Enter your choice (1-6): ");
}

// Main function
int main(int argc, char *argv[]) {
    const char* input_source = NULL;
    char* file_content = NULL;
    int run_test_suite = 0;
    int use_console_input = 0;
    int interactive_mode = argc == 1; // If no arguments are provided, go to interactive mode
    char filename[256];

    printf("Recursive Descent Parser\n");
    printf("Default LTD value: %d\n", g_student_ltd_value);

    if (!interactive_mode) {
        printf("Usage: %s [-ltd NUM] [-test] [-console] [-interactive] [filename]\n", argv[0]);
        printf("  -ltd NUM     : Set custom Last Three Digits value\n");
        printf("  -test        : Run the test suite\n");
        printf("  -console     : Read input from console\n");
        printf("  -interactive : Show interactive menu\n");
        printf("  filename     : Read input from specified file\n\n");
    }

    // Parse command line arguments if not in interactive mode
    int arg_offset = 1;
    while (!interactive_mode && arg_offset < argc) {
        if (strcmp(argv[arg_offset], "-ltd") == 0 && arg_offset + 1 < argc) {
            g_student_ltd_value = atoi(argv[arg_offset + 1]);
            printf("Using custom LTD value from command line: %d\n", g_student_ltd_value);
            arg_offset += 2;
        } else if (strcmp(argv[arg_offset], "-test") == 0) {
            run_test_suite = 1;
            arg_offset++;
        } else if (strcmp(argv[arg_offset], "-console") == 0) {
            use_console_input = 1;
            arg_offset++;
        } else if (strcmp(argv[arg_offset], "-interactive") == 0) {
            interactive_mode = 1;
            arg_offset++;
        } else {
            break;
        }
    }

    // Interactive menu handling
    if (interactive_mode) {
        int choice;
        do {
            display_interactive_menu();
            if (scanf("%d", &choice) != 1) {
                // Clear input buffer if scanf fails
                int c;
                while ((c = getchar()) != '\n' && c != EOF);
                choice = 0;  // Invalid choice
            }

            // Clear any remaining characters in input buffer
```

```
802             while (getchar() != '\n');
803
804             switch (choice) {
805                 case 1: // Console input
806                     printf("Enter your code (end with Ctrl+D on Unix/Linux or Ctrl+Z+Enter
    on Windows):\n");
807                     file_content = read_from_console();
808                     if (file_content) {
809                         input_source = file_content;
810
811                         printf("\nParsing the following input:\n---\n%s\n---\n\n",
    input_source);
812                         initialize_parser(input_source);
813
814                         // Try to parse with error handling
815                         jmp_buf env;
816                         if (setjmp(env) == 0) {
817                             program();
818                             printf("\n-------------------------------------\n");
819                             printf("Program parsed successfully!\n");
820                             printf("-------------------------------------\n");
821                         } else {
822                             printf("\n-------------------------------------\n");
823                             printf("Parsing failed!\n");
824                             printf("-------------------------------------\n");
825                         }
826
827                         free(file_content);
828                         file_content = NULL;
829                     }
830                     break;
831
832                 case 2: // File input
833                     printf("Enter filename: ");
834                     if (scanf("%255s", filename) == 1) {
835                         file_content = read_file_to_string(filename);
836                         if (file_content) {
837                             input_source = file_content;
838
839                             printf("\nParsing file: %s\n", filename);
840                             printf("---\n%s\n---\n\n", input_source);
841                             initialize_parser(input_source);
842
843                             // Try to parse with error handling
844                             jmp_buf env;
845                             if (setjmp(env) == 0) {
846                                 program();
847                                 printf("\n-------------------------------------\n");
848                                 printf("Program parsed successfully!\n");
849                                 printf("-------------------------------------\n");
850                             } else {
851                                 printf("\n-------------------------------------\n");
852                                 printf("Parsing failed!\n");
853                                 printf("-------------------------------------\n");
854                             }
855
856                             free(file_content);
857                             file_content = NULL;
858                         } else {
859                             printf("Error: Could not read file '%s'\n", filename);
860                         }
861                     }
862                     break;
863
864                 case 3: // Test suite
865                     {
866                         printf("Running test suite...\n");
867
868                         const int valid_test_count = 4;  // First 4 test cases are valid
869                         const int total_test_count = sizeof(test_cases) /
    sizeof(test_cases[0]);
870
871                         for (int i = 0; i < total_test_count; i++) {
872                             process_test_case(test_cases[i], i + 1, i < valid_test_count);
873                         }
874
875                         printf("\nTest suite completed.\n");
876                     }
877                     break;
878
879                 case 4: // Default test case
880                     input_source = test_cases[0];
881
882                     printf("\nParsing default test case:\n---\n%s\n---\n\n", input_source);
```

```
                    initialize_parser(input_source);

                    // Try to parse with error handling
                    {
                        jmp_buf env;
                        if (setjmp(env) == 0) {
                            program();
                            printf("\n-----------------------------------\n");
                            printf("Program parsed successfully!\n");
                            printf("-----------------------------------\n");
                        } else {
                            printf("\n-----------------------------------\n");
                            printf("Parsing failed!\n");
                            printf("-----------------------------------\n");
                        }
                    }
                    break;

                case 5: // Change LTD value
                    printf("Current LTD value: %d\n", g_student_ltd_value);
                    printf("Enter new LTD value: ");
                    int new_ltd;
                    if (scanf("%d", &new_ltd) == 1) {
                        g_student_ltd_value = new_ltd;
                        printf("LTD value updated to: %d\n", g_student_ltd_value);
                    } else {
                        printf("Invalid input. LTD value unchanged.\n");
                        // Clear input buffer
                        int c;
                        while ((c = getchar()) != '\n' && c != EOF);
                    }
                    break;

                case 6: // Exit
                    printf("Exiting program. Goodbye!\n");
                    break;

                default:
                    printf("Invalid choice. Please enter a number between 1 and 6.\n");
            }

            if (choice != 6) {
                printf("\nPress Enter to continue...");
                getchar();  // Wait for user to press enter
            }

        } while (choice != 6);

        return 0;
    }


    // Process based on provided flags
    if (run_test_suite) {
        printf("Running test suite...\n");

        // Count of valid test cases (the first 4)
        const int valid_test_count = 4;
        const int total_test_count = sizeof(test_cases) / sizeof(test_cases[0]);

        for (int i = 0; i < total_test_count; i++) {
            process_test_case(test_cases[i], i + 1, i < valid_test_count);
        }

        printf("\nTest suite completed.\n");
        return 0;
    }

    // Get input source (priority: console > file > default test case)
    if (use_console_input) {
        printf("Reading from console input...\n");
        file_content = read_from_console();
        if (!file_content) {
            return 1; // Error reading from console
        }
        input_source = file_content;
    }
    else if (argc > arg_offset) { // A filename is provided
        printf("Attempting to read input from file: %s\n", argv[arg_offset]);
        file_content = read_file_to_string(argv[arg_offset]);
        if (!file_content) {
            return 1; // Error reading file
        }
        input_source = file_content;
```

```
967         }
968         else {
969             // Default test case if no file is provided
970             printf("No input file provided. Using a default valid test case.\n");
971             input_source = test_cases[0]; // Use first test case as default
972         }
973 
974         printf("\nParsing the following input:\n---\n%s\n---\n\n", input_source);
975 
976         initialize_parser(input_source);
977         program(); // Start parsing
978 
979         printf("\n-------------------------------------\n");
980         printf("Program parsed successfully!\n");
981         printf("-------------------------------------\n");
982 
983         if (file_content) {
984             free(file_content); // Clean up if content was read from file
985         }
986 
987         return 0; // Success
988     }
```