```
std::string Expr::to_string() {
    std::stringstream st("");
    this->printExp(st);
    return st.str(); }
std::string Expr::to_pretty_string() {
    std::stringstream st;
    this->pretty_print(st, prec_none);
    return st.str(); }
void Expr::pretty_print_at(std::ostream& ot, precedence_t prec,
std::streampos& last_newline_pos) { printExp(ot); }
void Expr::pretty_print(std::ostream& ot) {
    std::streampos last_newline_pos = ot.tellp();
    pretty_print_at(ot, prec, last_newline_pos); }
NumExpr::NumExpr(int value) { this->value = value; }
int NumExpr::interp() { return value; }
bool NumExpr::has_variable() { return false; }
Expr* NumExpr::subst(const std::string& var, Expr* replacement) {
    return this; }
bool NumExpr::equals(const Expr* e) {
    const NumExpr* numExpr = dynamic_cast<const NumExpr*>(e);
    return numExpr && this->value == numExpr->value; }
void NumExpr::printExp(std::ostream& ot) { ot << value; }
AddExpr::AddExpr(Expr* lhs, Expr* rhs) {
    this->lhs = lhs;
    this->rhs = rhs; }
int AddExpr::interp() { return lhs->interp() + rhs->interp(); }
bool AddExpr::has_variable() {
    return lhs->has_variable() || rhs->has_variable(); }
Expr* AddExpr::subst(const std::string& var, Expr* replacement) {
    return new AddExpr(lhs->subst(var, replacement), rhs->subst(var, replacement)); }
bool AddExpr::equals(const Expr* e) {
    const AddExpr* addExpr = dynamic_cast<const AddExpr*>(e);
    return addExpr && lhs->equals(addExpr->lhs) && rhs->equals(addExpr->rhs); }
void AddExpr::printExp(std::ostream& ot) {
    ot << "(";
    lhs->printExp(ot);
    ot << "+";
    rhs->printExp(ot);
    ot << ")"; }
void AddExpr::pretty_print_at(std::ostream& ot, precedence_t prec, std::streampos&
last_newline_pos) {
    bool use_parentheses = (prec >= prec_add);
    if (use_parentheses) { ot << "("; }
    lhs->pretty_print_at(ot, prec_add, last_newline_pos);
    ot << " + ";
    rhs->pretty_print_at(ot, prec_none, last_newline_pos);
    if (use_parentheses) { ot << ")"; } }
MultExpr::MultExpr(Expr* lhs, Expr* rhs) {
    this->lhs = lhs;
    this->rhs = rhs; }
int MultExpr::interp() { return lhs->interp() * rhs->interp(); }
bool MultExpr::has_variable() {
    return lhs->has_variable() || rhs->has_variable(); }
Expr* MultExpr::subst(const std::string& var, Expr* replacement) {
    return new MultExpr(lhs->subst(var, replacement), rhs->subst(var, replacement)); }
bool MultExpr::equals(const Expr* e) {
    const MultExpr* multExpr = dynamic_cast<const MultExpr*>(e);
    return multExpr && lhs->equals(multExpr->lhs) && rhs->equals(multExpr->rhs); }
void MultExpr::printExp(std::ostream& ot) {
    ot << "(";
    lhs->printExp(ot);
    ot << "*";
    rhs->printExp(ot);
    ot << ")"; }
void MultExpr::pretty_print_at(std::ostream& ot, precedence_t prec, std::streampos&
last_newline_pos) {
    bool use_parentheses = (prec >= prec_mult);
    if (use_parentheses) { ot << "("; }
    lhs->pretty_print_at(ot, prec_mult, last_newline_pos);
    ot << " * ";
    rhs->pretty_print_at(ot, prec_add, last_newline_pos);
    if (use_parentheses) { ot << ")"; } }
VarExpr::VarExpr(const std::string& name) {
    this->name = name; }
int VarExpr::interp() { throw std::runtime_error("Variable has no value"); }
bool VarExpr::has_variable() { return true; }
Expr* VarExpr::subst(const std::string& var, Expr* replacement) {
    if (this->name == var) { return replacement; }
    return this; }
bool VarExpr::equals(const Expr* e) {
    const VarExpr* varExpr = dynamic_cast<const VarExpr*>(e);
    return varExpr && this->name == varExpr->name; }
void VarExpr::printExp(std::ostream& ot) { ot << name; }
LetExpr::LetExpr(const std::string& var, Expr* rhs, Expr* body) {
    this->var = var;
    this->rhs = rhs;
    this->body = body; }
bool LetExpr::equals(const Expr* e) {
    const LetExpr* letExpr = dynamic_cast<const LetExpr*>(e);
    return letExpr && this->var == letExpr->var &&
           this->rhs->equals(letExpr->rhs) &&
           this->body->equals(letExpr->body); }
int LetExpr::interp() {
    int rhsValue = rhs->interp();
    Expr* substitutedBody = body->subst(var, new NumExpr(rhsValue));
    return substitutedBody->interp(); }
bool LetExpr::has_variable() {
    return rhs->has_variable() || body->has_variable(); }
Expr* LetExpr::subst(const std::string& var, Expr* replacement) {
    if (this->var == var) {
        return new LetExpr(this->var, rhs->subst(var, replacement), body); }
    return new LetExpr(this->var, rhs->subst(var, replacement), body->subst(var,
replacement)); }
void LetExpr::printExp(std::ostream& ot) {
    ot << "(_let " << var << "=";
    rhs->printExp(ot);
    ot << " _in ";
    body->printExp(ot);
    ot << ")"; }
void LetExpr::pretty_print(std::ostream& ot, precedence_t prec) {
    std::streampos last_newline_pos = ot.tellp();
    pretty_print_at(ot, prec, last_newline_pos); }
void LetExpr::pretty_print_at(std::ostream& ot, precedence_t prec, std::streampos&
last_newline_pos) {
    bool needs_parentheses = (prec != prec_none);
    if (needs_parentheses) { ot << "("; }
    std::streampos position1 = last_newline_pos;
    std::streampos current_pos = ot.tellp();
    ot << "_let " << var << " = ";
    rhs->pretty_print(ot, prec_none);
    ot << "\n";
    last_newline_pos = ot.tellp();
    for (int i = position1; i < current_pos; i++) {
        ot << " "; }
    ot << "_in  ";
    body->pretty_print_at(ot, prec_none, last_newline_pos);
    if (needs_parentheses) { ot << ")"; } }
```

typedef enum{
  do_nothing,
  do_test,
  do_interp,
  do_print,
  do_pretty_print
} run_mode_t;

```
// Purpose: Parses command-line arguments and returns the corresponding run mode.
// Parameters:
//   - argc: The number of command-line arguments.
//   - argv: An array of C-style strings representing the command-line arguments.
// Returns: A run_mode_t value indicating the mode of operation (e.g., do_test,
do_interp..
```

CmdLine.cpp

```
run_mode_t use_arguments(int argc, char **argv) {
    // Check if the number of arguments is not equal to 2.
    // The program expects exactly 2 arguments: the program name and a flag.
    if (argc != 2) {
        // Print an error message to standard error if the number of arguments is incorrect.
        std::cerr << "Usage: msdscript [--test | --interp | --print | --pretty-print]\n";
        exit(1); } // Exit the program with a non-zero status code (1) to indicate an error.
    // Extract the second argument (the flag) from the argv array.
    // argv[0] is the program name, and argv[1] is the flag.
    std::string flag = argv[1];
    // Check the value of the flag and return the corresponding run mode.
    if (flag == "--test") {
        // If the flag is "--test", return do_test to indicate test mode.
        return do_test;
    } else if (flag == "--interp") { return do_interp; }
      else if (flag == "--print") { return do_print; }
      else if (flag == "--pretty-print") { return do_pretty_print; }
      else {
        // If the flag is not recognized, print an error message to standard error.
        std::cerr << "Invalid flag. Use --test, --interp, --print, or --pretty-print\n";
        exit(1); } } // Exit the program with a non-zero status code (1) to indicate an error.
// Define the CATCH_CONFIG_RUNNER macro to enable the Catch2 test framework.
// This allows the program to run tests when the --test flag is provided.
#define CATCH_CONFIG_RUNNER
int main(int argc, char* argv[]) {
    try { // Parse command-line arguments and determine the run mode
        run_mode_t mode = use_arguments(argc, argv);
        if (mode == do_test) { // If the mode is do_test, run the Catch2 test suite
            int result = Catch::Session().run();  // Create a Catch2 session and run the tests
            return result; } // Return the test result (0 for success, non-zero for failures)
        std::string input;
        std::getline(std::cin, input); // Read input from standard input
        std::stringstream ss(input); // Create a string stream from the input for parsing
        Expr* expr = parse_expr(ss); // Parse the input expression into an Expr object
        switch (mode) { // Handle the flag based on the run mode
            case do_interp: {
                // If the mode is do_interp, interpret the expression and print the result
                int result = expr->interp();
                std::cout << result << "\n";
                break; }
            case do_print: { // If the mode is do_print, print the expression as a string
                std::cout << expr->to_string() << "\n";
                break; }
            case do_pretty_print: {
                // If the mode is do_pretty_print, pretty-print the expression
                std::cout << expr->to_pretty_string() << "\n";
                break; }
            default:
                std::cerr << "Invalid mode.\n"; // If the mode is invalid, print an error message
                return 1; } // Exit with a non-zero status code to indicate an error
                return 0;// Exit with a status code of 0 to indicate success
    } catch (const std::exception& e) {
        // Catch any exceptions that occur during execution
        // Print the error message to standard error
        std::cerr << "Error: " << e.what() << "\n";
        return 1; } } // Exit with a non-zero status code to indicate an error
```

Main.cpp

```
    # Compiler and compiler flags
CXX = g++ # Use g++ as the C++ compiler
CXXFLAGS = -Wall -Wextra -std=c++17  # Compiler flags:
# -Wall: Enable all warnings
# -Wextra: Enable extra warnings
# -std=c++17: Use the C++17 standard
# Target executables
TARGET = msdscript # Name of the main executable
TEST_TARGET = test_msdscript  # Name of the test executable
# Source and object files for the main program
SRCS = main.cpp expr.cpp cmdline.cpp tests.cpp parse.cpp  # List of
source files
OBJS = $(SRCS:.cpp=.o) # Generate object file names by replacing .cpp
with .o
# Source and object files for the test program
TEST_SRCS = test_msdscript.cpp exec.cpp  # List of source files for the
test program
TEST_OBJS = $(TEST_SRCS:.cpp=.o)  # Generate object file names by
replacing .cpp with .o
all: $(TARGET) # Default target: build the main executable
# Rule to build the main executable
$(TARGET): $(OBJS) # The target depends on the object files
    $(CXX) $(CXXFLAGS) -o $@ $^  # Link the object files into the executable
# $@: The target (msdscript)
# $^: All dependencies (object files)
# Rule to build the test executable
$(TEST_TARGET): $(TEST_OBJS)  # The target depends on the test object
files
    $(CXX) $(CXXFLAGS) -o $@ $^  # Link the test object files into the test
executable
# $@: The target (test_msdscript)
# $^: All dependencies (test object files)
clean:
    rm -f $(OBJS) $(TARGET)    # Remove object files and the main
executable
# -f: Force removal (ignore errors if files don't exist)
# Phony targets (targets that are not actual files)
.PHONY: all clean test
# Target to run tests
test: $(TARGET) # The test target depends on the main executable
    ./$(TARGET) --test # Run the main executable with the --test flag
# Target to generate documentation
doc: msdscript # The doc target depends on the main executable
    cd documentation && doxygen  # Change to the documentation directory
and run Doxygen
```

```
// Purpose: Consumes a specific character from the input stream.
//   - expect: The expected character to consume.
// Throws: std::runtime_error if the next character does not match the expected
character.
static void consume(std::istream& in, int expect) {
    int c = in.get();  // Read the next character from the input stream
    if (c != expect) { // Check if the character matches the expected one
        throw std::runtime_error("bad input"); } } // Throw an error if it doesn't match
void skip_whitespace(std::istream& in) {
    while (true) {
        int c = in.peek(); // Peek at the next character without consuming it
        if (!isspace(c)) break; // Exit the loop if the character is not whitespace
        consume(in, c); // Consume the whitespace character } }
// Purpose: Parses a number (integer) from the input stream.
// Returns: A pointer to a NumExpr object representing the parsed number.
// Throws: std::runtime_error if the input is invalid (e.g., no digit after '-').
Expr* parse_num(std::istream& in) {
    int n = 0;          // Initialize the number to 0
    bool negative = false; // Flag to indicate if the number is negative
    if (in.peek() == '-') {  // Check if the number is negative
        negative = true; // Set the negative flag
        consume(in, '-'); // Consume the '-' character
        if (!isdigit(in.peek())) {  // Ensure there is a digit after the '-'
            throw std::runtime_error("invalid input"); } }// Throw an error if no digit
follows
    while (true) {   // Parse the digits of the number
        int c = in.peek(); // Peek at the next character
        if (isdigit(c)) {  // If it's a digit, consume it and add to the number
            consume(in, c);
            n = n * 10 + (c - '0'); // Convert the character to a digit and add to the
number
        } else { break;} } // Exit the loop if a non-digit character is encountered
    if (negative) {     // Apply the negative sign if necessary
        n = -n; }
    return new NumExpr(n);} // Return a new NumExpr object representing the
parsed number
// Purpose: Parses a variable name from the input stream.
// Throws: std::runtime_error if the variable name contains invalid characters (e.g.,
' ').
Expr* parse_var(std::istream& in) {
    std::string name; // Initialize an empty string to store the variable name
    while (true) {
        int c = in.peek(); // Peek at the next character
        if (isalpha(c)) {  // If it's an alphabetic character, consume it and add to the
name
            consume(in, c);
            name += static_cast<char>(c);
        } else if (c == '_') {          // If an underscore is encountered, throw an
exception
            throw std::runtime_error("invalid input");
        } else { break;} }// Exit the loop if a non-alphabetic character is encountered
    if (name.empty()) {   // Ensure the variable name is not empty
        throw std::runtime_error("invalid input"); }
// Return a new VarExpr object representing the parsed variable
    return new VarExpr(name); }
// Purpose: Parses a multicand (number, variable, or parenthesized expression).
// Throws: std::runtime_error if the input is invalid.
Expr* parse_multicand(std::istream& in) {
    skip_whitespace(in); // Skip any leading whitespace
    int c = in.peek();   // Peek at the next character
    if ((c == '-') || isdigit(c)) {   // Handle numbers (including negative numbers)
        return parse_num(in); }
    else if (c == '(') {   // Handle parenthesized expressions
        consume(in, '('); // Consume the '(' character
        Expr* e = parse_expr(in); // Parse the inner expression
        skip_whitespace(in); // Skip any trailing whitespace
        consume(in, ')'); // Consume the ')' character
        return e;}
    else if (isalpha(c)) {   // Handle variables
        return parse_var(in);}
    else {    // Handle invalid input
        throw std::runtime_error("bad input");} }
// Purpose: Parses an addend (multicand or multiplication expression).
Expr* parse_addend(std::istream& in) {
    Expr* lhs = parse_multicand(in); // Parse the left-hand side of the multiplication
    skip_whitespace(in); // Skip any whitespace
    int c = in.peek();   // Peek at the next character
    if (c == '*') {    // Handle multiplication
        consume(in, '*'); // Consume the '*' character
        Expr* rhs = parse_addend(in); // Parse the right-hand side of the multiplication
        return new MultExpr(lhs, rhs);} // Return a new MultExpr object
    return lhs;}    // If no multiplication, return the left-hand side
// Purpose: Parses an expression (addend or addition expression).
// Returns: A pointer to an Expr object representing the parsed expression.
Expr* parse_expr(std::istream& in) {
    Expr* lhs = parse_addend(in); // Parse the left-hand side of the addition
    skip_whitespace(in); // Skip any whitespace
    int c = in.peek();   // Peek at the next character
    if (c == '+') {    // Handle addition
        consume(in, '+'); // Consume the '+' character
        Expr* rhs = parse_expr(in); // Parse the right-hand side of the addition
        return new AddExpr(lhs, rhs);} // Return a new AddExpr object
    return lhs;}    // If no addition, return the left-hand side
// Purpose: Main parse function that parses an expression from the input stream.
// Returns: A pointer to an Expr object representing the parsed expression.
Expr* parse(std::istream& in) {
    return parse_expr(in);} // Delegate to parse_expr
// Purpose: Wrapper for testing that parses a string into an expression.
Expr* parse_str(const std::string& s) {
    std::stringstream ss(s); // Create a string stream from the input string
    return parse(ss);} // Parse the expression from the string stream
```