FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

University of West Bohemia

Faculty of Applied Sciences

KIV/PC Programming in C

Semester Project

Graph Visualization of Mathematical Functions

Student: Malakhov Leonid
Student ID: A22B0387P
Course: KIV/PC Programming in C
Date: January 4, 2025

1 Task Description

The task is to develop a portable console application in ANSI C that takes a mathematical function as input, analyzes it, and generates a PostScript file with the graph of the function over a specified domain. The program should be executed with the following command:

```
graph.exe <func> <out-file> [<limits>]
```

Where:

- `<func>` represents the mathematical function, where `x` is the independent variable and `y` is the dependent variable (implicitly declared). The function should be in a form such as `y = f(x)` with one independent variable `x`.
- `<out-file>` is the name of the output PostScript file.
- `<limits>` (optional) specifies the display range of the graph, including the domain and range for the function.

The program must handle basic mathematical functions (e.g., sine, cosine, logarithms) and operators (e.g., addition, subtraction, multiplication, division, exponentiation). It must also be able to interpret various representations of the function, including with or without spaces and quotation marks.

The input should be provided entirely through command-line arguments, and there should be no interaction with the user during the execution of the program.

The program will generate a PostScript file displaying the graph with labeled axes and relevant points of the function's domain and range.

2 Problem Analysis

2.1 Problem 1: Converting the Text Argument of the Program in the Form of a Function into a Format Recognizable by the Computer

2.1.1 Definition

When the user enters a function as a text string (e.g., ”x + 2 + 2*x - 3”), it needs to be transformed into a structure that the program can interpret and analyze. This task includes several sub-tasks:

- **Recognition of Symbols and Expressions:** The textual representation of the function may contain various symbols (e.g., operators like +, -, *, /, parentheses, variables like x, numbers, etc.). All these symbols must be correctly interpreted.
- **Conversion into Data Structure:** For further use of the mathematical function, the text must be converted into corresponding data structures, such as a tree or an array, which will be convenient for calculations.
- **Handling Mathematical Operations:** It is necessary to account for the fact that operations may have different priorities (for example, multiplication and division are performed before addition and subtraction). This requires the proper handling of the order of operations.

2.1.2 Solutions

Several possible approaches can be used to convert a text-based mathematical function into a format that the program can process:

- **Regular Expressions:** Using regular expressions to match and parse the input string. This approach can handle basic mathematical expressions but may struggle with more complex cases (e.g., nested parentheses or different types of operators).
- **Reverse Polish Notation and Shunting Yard Algorithm:** The input string can be tokenized into individual symbols (numbers, variables, operators, parentheses) and then processed using the Shunting Yard algorithm, which converts the infix notation to a form suitable for computation, such as Reverse Polish Notation (RPN). For the expression

  $$4 + (2 * 10 + 3 * (5 + 1))$$

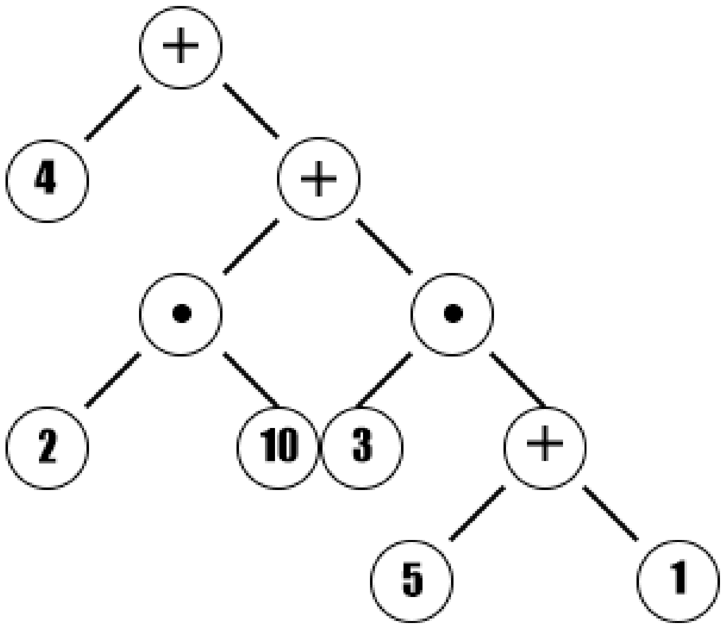
  Reverse Polish Notation will look like this:

  $$4 2 10 * 3 5 1 + * + +$$

- **Abstract Syntax Tree (AST):** The function can be converted directly into an abstract syntax tree. This involves parsing the input into hierarchical nodes, each representing a mathematical operation or operand. This is a more robust solution for complex expressions. For the expression

$$4 + (2 * 10 + 3 * (5 + 1))$$

AST will look like on the picture:



- **Parser Generators:** Using parser generators like ANTLR or Bison to create a parser specifically for mathematical expressions. These tools can handle complex syntax rules and generate the corresponding code for parsing and interpreting mathematical functions.

2.1.3 Chosen Solution

For solving this problem, I used **tokenization** and **abstract syntax tree (AST)**. First, the textual string is split into individual tokens, which are then converted into a syntax tree, allowing more convenient and accurate interpretation of the mathematical expression. This solution appeared clearer and simpler to understand, and it meets all the requirements of the task.

2.2 Problem 2: Checking the Correctness of the Function Syntax

2.2.1 Definition

Once the function has been transformed into a format recognizable by the program, it is necessary to check whether it is correctly written. This check involves several aspects:

- **Syntax Check:** It must be ensured that the entered string follows the syntax of mathematical expressions, e.g., that all parentheses are correctly placed, operators do not follow each other incorrectly (e.g., "++" or "**"), and there are no extraneous symbols.
- **Check of Valid Operations and Variables:** It is necessary to ensure that all operations are valid for the variables and numbers used (for example, dividing by zero or using an undefined variable is not allowed).
- **Expression Structure Check:** It is important to ensure that the expression has a valid structure: there should be no extraneous operations or incorrect order of operations.

2.2.2 Solutions

To check if the function syntax is correct, several strategies can be employed:

- **Grammar Checking with Regular Expressions:** Regular expressions can be used to check basic syntax patterns, such as ensuring that parentheses are properly matched, operators do not follow each other incorrectly, and there are no extraneous symbols.
- **Parse Tree Validation:** If the function is parsed into an abstract syntax tree, it can be validated by traversing the tree and checking for structural errors (e.g. invalid operator positions or undefined variables).
- **Mathematical Expression Validator:** A specialized validation algorithm could be written to specifically check for errors related to mathematical expressions, such as division by zero or invalid operator usage. This could be done as part of the parsing process.
- **Third-Party Libraries:** Existing libraries, such as those in mathematical expression parsers, can be used to validate the syntax and structure of mathematical functions.

2.2.3 Chosen Solution

For checking the correctness of the function, I used two approaches. First, during the **tokenization** phase, I check the symbols and the balance of parentheses to ensure that the syntax is correct. Second, when constructing the abstract syntax tree, I check the **sequence of tokens** to verify the correctness of the expression structure.

2.3 Problem 3: Computing the Value of the Function

2.3.1 Definition

Once the function is correctly formatted and checked for errors, the next step is to calculate its value for given input values (e.g., for $x = 2$).

- **Evaluating the Function Value for a Specific Argument:** The function must be correctly evaluated based on the provided arguments, considering the operations in the correct order.

2.3.2 Solutions

Once the function is properly formatted and validated, calculating its value for specific arguments is the next step. Possible solutions include:

- **Evaluator Function with AST Traversal:** If the function has been parsed into an abstract syntax tree, a recursive function can traverse the tree, applying the appropriate operations to compute the value based on the input values.
- **Reverse Polish Notation (RPN) Evaluation:** If the function is converted into Reverse Polish Notation (RPN) using the shunting yard algorithm, an evaluator can then process the RPN expression using a stack to calculate the result.

2.3.3 Chosen Solution

To compute the function value, I use **recursive evaluation of the abstract syntax tree**. For each value of the variable x, the tree is traversed recursively, substituting values and performing operations to compute the final result.

2.4 Problem 4: Drawing the Function Graph

2.4.1 Definition

The final step is visualizing the function, i.e., plotting its graph on the screen. This requires:

- **Defining the Range of Values:** It is necessary to determine which values of the variable (e.g., x) will be used for plotting the graph, and what range the graph should cover.
- **Plotting the Graph Points:** The value of the function needs to be computed for a series of variable values, and these points need to be plotted on the graph.
- **Defining the Correct Scales for the Axes:** Proper scaling for the coordinate axes must be determined, and the graph needs to be adapted to the chosen range of values.

- **Displaying the Graph:** Once the points are calculated, they need to be connected to form the graph. It is also important to properly configure the coordinate axes, value displays, grid, and other elements to enhance the perception of the graph.

2.4.2 Solutions

To visualize the function by plotting its graph, there are several approaches that can be taken:

- **Plotting Using Discrete Points:** Compute the function value for a range of x-values and plot the resulting points on a Cartesian coordinate system.

2.4.3 Chosen Solution

For plotting the function graph, I calculate the y-values for each x, and then I **draw lines between the points**. This approach allows for an accurate representation of the function graph by connecting adjacent points, thus creating the required visual output.

3 Implementation Description

3.1 Main Processes of the Program (Happy Day Scenario)

The program follows a sequential process to achieve its goal of generating a graph of a mathematical function. The main steps are described below:

1. **Initialization of Limits:**
   - `limits = initialize_limits();`
   - This function initializes the `Limits` structure with default values for the x and y axes.
2. **Parsing Limits:**
   - `parse_limits(argv[3], limits);`
   - If the user provides specific limits as a command-line argument, this function parses the input and updates the `Limits` structure accordingly.
3. **Opening the Output File:**
   - `output_file = fopen(output_file_name, "w");`
   - The output file is opened for writing the generated graph. If the file cannot be created or opened, the program will terminate with an error.
4. **Lexer Initialization:**
   - `lexer = initialize_lexer(expression);`
   - The lexer is initialized with the mathematical expression provided as input. It prepares for tokenization and parsing.
5. **Abstract Syntax Tree (AST) Construction:**
   - `abstract_syntax_tree = parse(lexer);`
   - The parser builds an Abstract Syntax Tree (AST) from the tokens produced by the lexer. The AST represents the mathematical expression in a structured format suitable for evaluation.
6. **Graph Generation:**
   - `draw_graph(limits, output_file, abstract_syntax_tree);`
   - The graph is generated based on the AST and the defined limits. It includes axes, gridlines, and the function plot, which is written to the output file.
7. **Cleanup:**
   - `cleanup();`
   - All dynamically allocated memory is freed, and the program ensures no memory leaks occur before terminating.

3.2 Program Structures and Variables

The program uses several global variables and structures, which play an essential role in managing its execution. Below is a detailed description of the main variables and the structures used throughout the program.

3.2.1 Global Variables

`static Limits *limits;` **Pointer to the graph limits.** This variable holds a structure defining the range of values for the graph axes (x and y). It ensures that the graph is rendered within the specified coordinate space.

`static FILE *output_file;` **Pointer to the output file.** This variable stores a pointer to the file where the graph or evaluation results will be written. The file format is typically PostScript (.ps).

`static Lexer *lexer;` **Pointer to the lexer.** The lexer is responsible for tokenizing the input mathematical expression into manageable units (tokens). This pointer maintains the lexer state throughout the program's execution.

`static Node *abstract_syntax_tree;` **Pointer to the Abstract Syntax Tree (AST).** The AST represents the parsed structure of the mathematical expression, where each node corresponds to an operator, operand, or function.

3.2.2 Structures

Lexer Structure

```
typedef struct Lexer {
    const char *text;  /* Input string to be tokenized. */
    size_t pos;        /* Current position in the input text. */
    char current_char; /* Current character being processed. */
} Lexer;
```

This structure holds the lexer state, including the input text, the current position, and the current character being processed.

Token Structure

```
typedef enum TokenType {
    TOKEN_NUM, TOKEN_ID, TOKEN_FUNC,
    TOKEN_PLUS, TOKEN_MINUS, TOKEN_MUL, TOKEN_DIV,
    TOKEN_LPAREN, TOKEN_RPAREN, TOKEN_POW, TOKEN_ERROR
} TokenType;

typedef struct Token {
    TokenType type;    /* Type of the token. */
    union {
        double num;    /* Numeric value if the token is a number. */
        char id[MAX_IDENTIFIER_LENGTH]; /* Identifier or function name. */
    };
} Token;
```

The `Token` structure represents a single token with its type and value, determined by the lexer.

Limits Structure

```
typedef struct Limits {
    double x_min; /* Minimum value for the x-axis. */
    double x_max; /* Maximum value for the x-axis. */
    double y_min; /* Minimum value for the y-axis. */
    double y_max; /* Maximum value for the y-axis. */
} Limits;
```

This structure defines the limits for the graph's x and y axes.

Node Structure (Abstract Syntax Tree)

```
typedef struct Node {
    enum type {
        NODE_NUM, NODE_ID, NODE_FUNC, NODE_OP, NODE_ERROR
    } type;

    union {
        double num; /* Numeric constant. */
        char id[MAX_IDENTIFIER_LENGTH]; /* Variable identifier. */
        struct {
            char func[MAX_IDENTIFIER_LENGTH]; /* Function name. */
            struct Node *arg; /* Argument node. */
        } func;
        struct {
            char op; /* Operator. */
            struct Node *left; /* Left operand node. */
            struct Node *right; /* Right operand node. */
        } op;
    };
} Node;
```

The `Node` structure represents elements of the AST, including numbers, variables, functions, and operators.

3.3 Program Modules

The program consists of several modules, each responsible for a specific part of the overall functionality. Every module includes two main files: a source file with the '.c' extension and a header file with the '.h' extension. The header files contain function declarations, constants defined using `define`, and documentation comments for each function. The following sections describe each module in detail.

3.3.1 Limits Module

The `limits` module is responsible for defining and parsing the limits for the x and y axes. The corresponding header file `limits.h` contains the structure definition for the `Limits` structure, which holds the values of the lower and upper bounds for both axes. The module also handles the initialization and parsing of these limits.

The functions for initializing and parsing the limits are declared in the header file and defined in `limits.c`.

3.3.2 Error Handling Module

The `err` module manages error handling throughout the program. The `err.h` header file contains error codes and corresponding error messages. The module provides functions for handling errors, including printing appropriate error messages.

3.3.3 Lexer Module

The `lexer` module is responsible for reading the mathematical function input and recognizing the different symbols used in the expression. The `lexer.h` header file contains `define` constants for all the symbols that the lexer can recognize, such as operators, parentheses, and numbers.

3.3.4 Parser Module

The `parser` module is responsible for constructing an abstract syntax tree (AST) based on the tokens generated by the lexer. The `parser.h` header file declares the functions used for building the tree, as well as the structure for representing nodes in the AST.

The parser works closely with the lexer to generate the abstract syntax tree, which is then used for evaluation in the `evaluator` module.

3.3.5 Evaluator Module

The `evaluator` module is responsible for evaluating the abstract syntax tree for a specific value of `x`. The `evaluator.h` header file contains the function declaration for evaluating the AST, which returns the result of the function at a given point.

The evaluator uses the tree structure provided by the parser to compute the value of the mathematical function at a specified point.

3.3.6 Draw Utils Module

The `draw_utils` module is responsible for generating the graphical output and saving it to a file. It includes functions for drawing axes, grid lines, and the function itself. The `draw_utils.h` header file contains constants for the drawing settings (e.g., axis colors, grid line spacing) and function declarations.

The draw utils module interacts with the evaluator to plot the function and its values on the generated graph.

3.3.7 Main Module

The `main` module is responsible for the program execution, including initializing the modules, reading input, and calling the appropriate functions in sequence. It also contains the `cleanup` function, which frees any dynamically allocated memory.

```
/* main.c */
#include "limits.h"
#include "err.h"
#include "lexer.h"
#include "parser.h"
#include "evaluator.h"
#include "draw_utils.h"

int main(int argc, char *argv[]) {
    Limits limits;
    char input[256];

    /* Initialize limits and read input */
    init_limits(&limits);
    get_input(input);

    /* Lexical analysis */
    Token tokens[256];
    lex_function(input, tokens);

    /* Parse the tokens into an abstract syntax tree */
    ASTNode* tree = parse_tokens(tokens);

    /* Evaluate the function for each x */
    FILE* file = fopen("output.ps", "w");
    draw_axes(file, &limits);
    draw_function(file, tree, &limits);
    fclose(file);

    /* Clean up */
    cleanup(tree);

    return 0;
}
```

The main module serves as the entry point for the program, handling the overall control flow and coordinating the work of the other modules.

3.4 Syntax Tree Construction

The syntax tree construction process is a core component of the program, enabling the parsing of mathematical expressions into a structured representation suitable for evaluation and visualization.

3.4.1 Initialization

The process begins with the initialization of the lexer using the function:

```
lexer = initialize_lexer(expression);
```

This function initializes a lexer that processes the input mathematical expression. During this step, the lexer performs an essential validation of the expression's brackets:

- It ensures that the number of opening and closing brackets matches.
- It verifies the correct order of brackets (e.g., a closing bracket cannot precede its corresponding opening bracket).

Bracket validation is handled by the function `are_brackets_balanced`:

```
int are_brackets_balanced(const char *expression);
```

If the brackets are not balanced, the program terminates with an error message.

3.4.2 Parsing the Expression

Once the lexer is initialized, the syntax tree is built by invoking the `parse` function:

```
abstract_syntax_tree = parse(lexer);
```

This function constructs the syntax tree recursively by parsing expressions with varying priorities (low and high) and operands. It ensures the entire expression is processed and raises an error if unexpected characters are found.

Parsing Functions The parsing process relies on a series of functions, each tailored to a specific level of operation priority:

- `parse_low_priority_expression`: Handles addition and subtraction.
- `parse_high_priority_expression`: Processes multiplication, division, and exponentiation.
- `parse_operand`: Parses basic operands such as numeric literals, identifiers, and sub-expressions in parentheses.

These functions create nodes in the syntax tree, with each node corresponding to an operator or operand derived from the lexer.

3.4.3 Lexer Operations

The lexer provides the tokens required for tree construction. Its key functions include:

- `advance`: Moves to the next character in the input.
- `skip_whitespace`: Skips over whitespace characters.
- `process_number`: Extracts numeric values, including integers and floating-point numbers.
- `process_identifier`: Identifies variables and functions.
- `process_operator`: Recognizes mathematical operators.
- `process_bracket`: Handles parentheses for sub-expressions.
- `get_next_token`: Retrieves the next token from the input expression.

These functions ensure the lexer accurately tokenizes the expression, enabling the parser to construct a valid syntax tree.

3.4.4 Recursive Tree Construction

The recursive nature of tree construction involves:

- Parsing low-priority operations, which recursively call functions for higher-priority operations.
- Building nodes for operators and operands as the recursion progresses.
- Ensuring correct handling of parentheses and operator precedence.

Each node in the syntax tree encapsulates an operation or operand, forming a hierarchical structure that represents the entire mathematical expression.

3.5 Graph Generation

3.5.1 Process Overview

The graph generation process begins with the main function call to the `draw_graph` function, which triggers a sequence of steps to render the graph in the PostScript format. These steps include calculating scaling factors, drawing axes, defining boundaries, adding grid lines, and finally drawing the mathematical function.

3.5.2 Preparing the Graph

The first step in the graph generation process is preparing the PostScript file for drawing the graph. This is done by the `prepare_graph` function, which sets up the page size, font, and coordinate system. It calculates the scaling factors for the x and y axes based on the provided limits and prepares the file for subsequent drawing operations.

3.5.3 Drawing the Axes

The next step involves drawing the coordinate axes. The function `draw_axes` is responsible for this task. It draws the X and Y axes, adds arrows at the positive ends, and labels the axes with 'x' and 'y'. The function positions the axes based on the scaling factors calculated earlier and ensures proper placement on the graph.

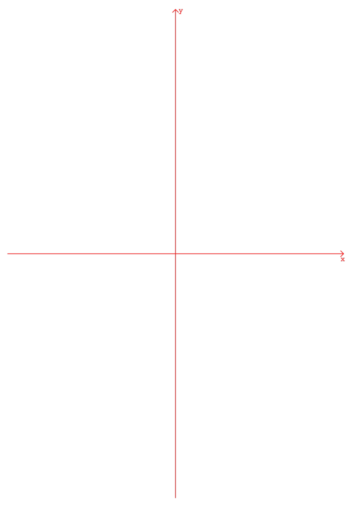


Figure 1: Graph of sine of x function with default limits after `draw_axes`.

3.5.4 Drawing the Limits

After the axes are drawn, the next step is to draw the boundary limits of the graph. This is achieved using the `draw_limits` function. The function draws dashed blue lines along the edges of the graph, indicating the boundaries for both the x and y axes. The limits are determined by the user-defined parameters in the `Limits` structure.

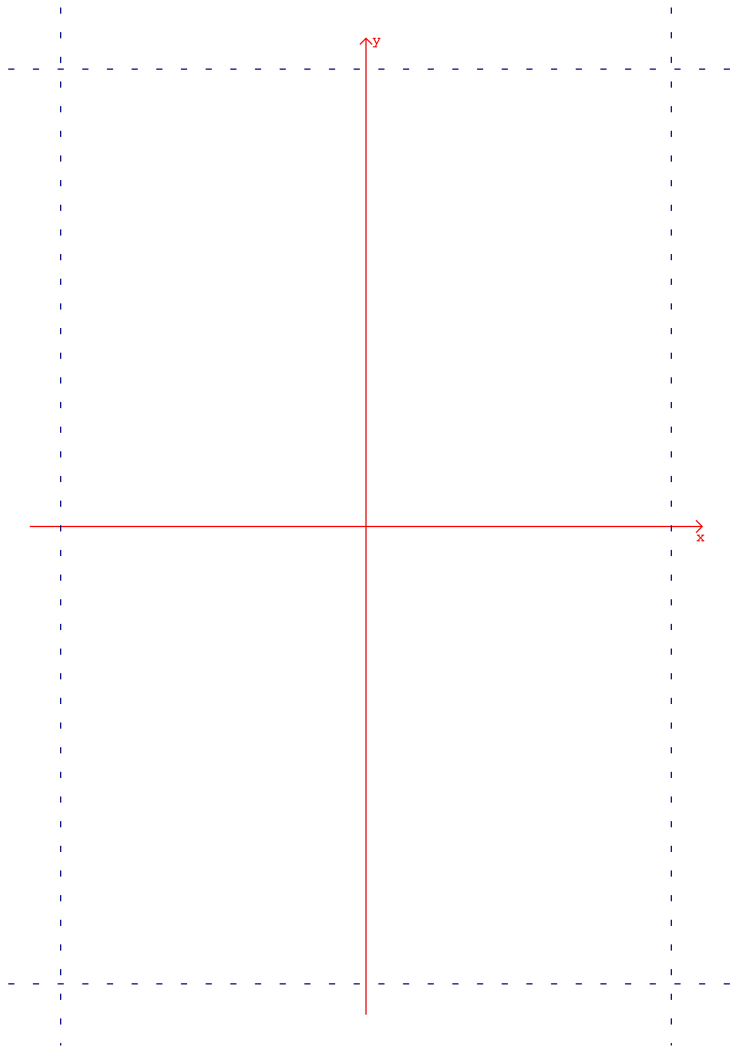


Figure 2: Graph of sine of x function with default limits after `draw_limits`.

3.5.5 Drawing the Support Lines

Next, supporting grid lines and tick marks are drawn on the axes to enhance the graph's readability. The function `draw_support_lines` is used for this purpose. It draws grid lines in grey and labels the tick marks with their corresponding values. This step also positions the tick marks based on the scaling factors and axis limits.

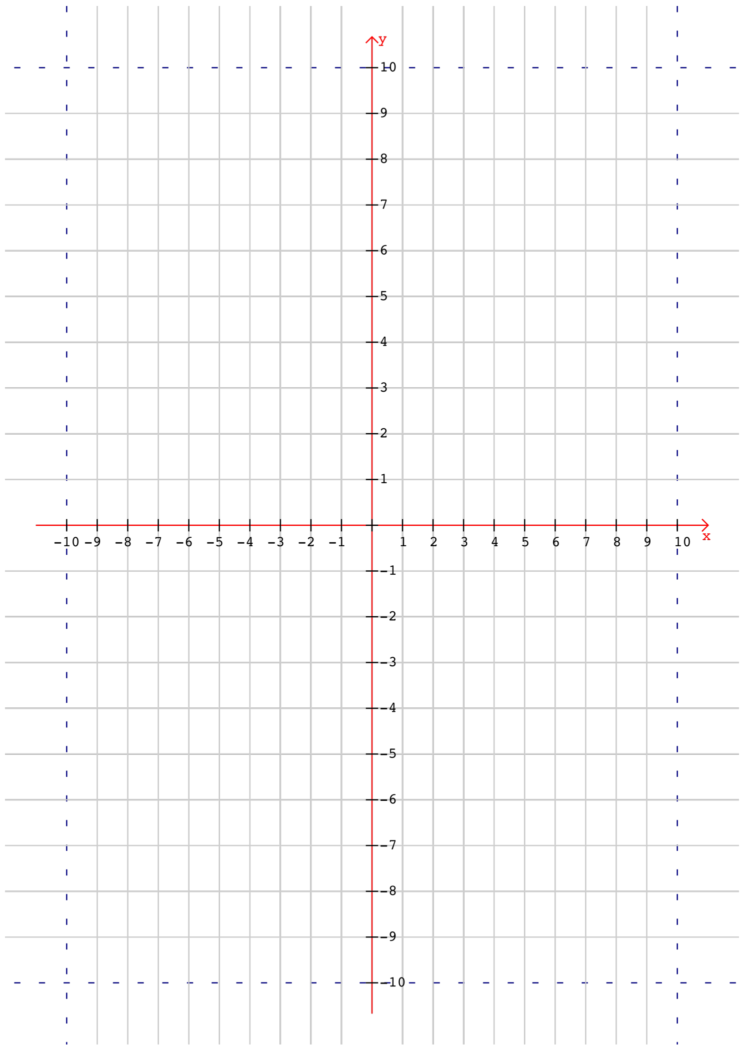


Figure 3: Graph of sine of x function with default limits after `draw_support_lines`.

3.5.6 Drawing the Function

Drawing the function is the most critical part of the graph generation process. This is handled by the `draw_function` function, which evaluates the mathematical function represented by the abstract syntax tree (AST) for a range of x-values within the specified limits. The function generates PostScript commands to plot the corresponding y-values on the graph.

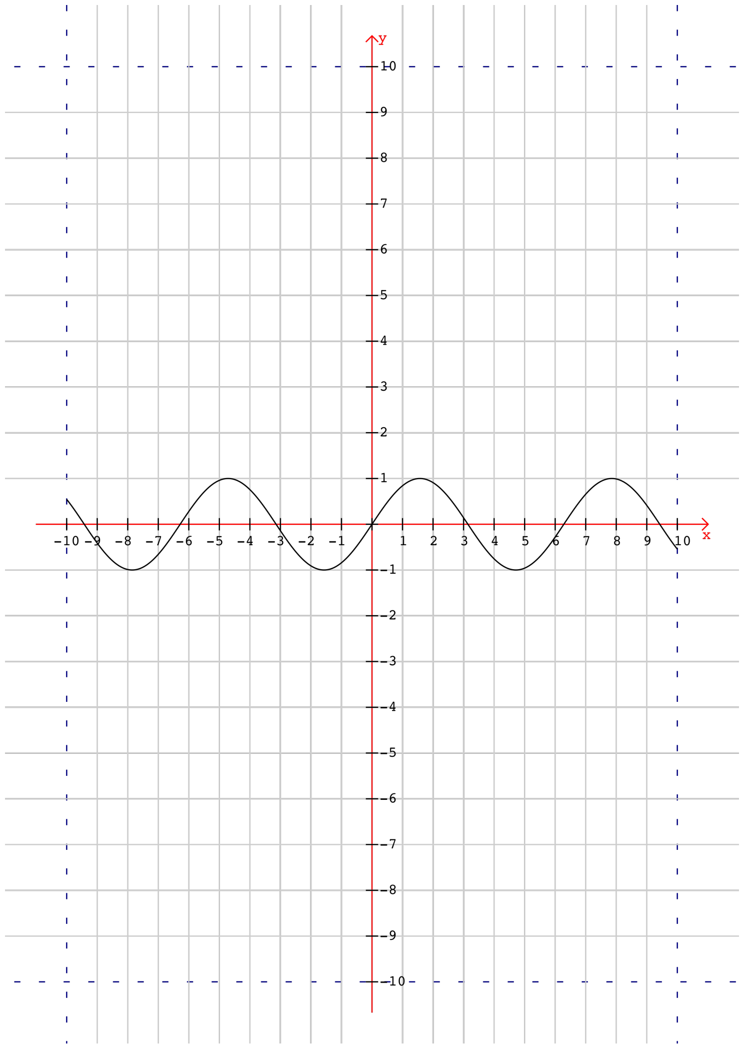


Figure 4: Graph of sine of x function with default limits after `draw_function`.

The process begins by iterating over the x-values within the limits defined by `x_min` and `x_max`. For each x, the `evaluate` function is called to calculate the corresponding y-value. The `evaluate` function traverses the abstract syntax tree and computes the value of the function for the given x. If the function encounters an invalid x-value (e.g., division by zero), it will skip that point and move to the next valid point.

Once a valid (x, y) pair is found, the function generates a PostScript command to either move to the point or draw a line to the new point, connecting consecutive points on the graph.

The `evaluate` function is responsible for calculating the value of the function at a specific x-value. It recursively traverses the abstract syntax tree, evaluates numbers, variables, and operators, and handles mathematical functions such as sine, cosine, and logarithms.

3.5.7 Finalizing the Drawing

After the function is drawn, the final step is to close the current drawing path and finalize the PostScript file. This is done by the `finish` function, which ensures the graph is properly rendered and the page is closed.

```
void finish(FILE *file);
```

3.6 Error Handling and Cleanup Functions

3.6.1 Error Exit Function

The function `error_exit` is used for handling errors in the program. When an irrecoverable error is encountered, it prints an error message to the standard error stream and terminates the program with the specified exit code.

```
void error_exit(const char *message, const int exit_code) {
    fprintf(stderr, "Error: %s.\n", message);
    exit(exit_code);
}
```

The `error_exit` function is typically called when an error cannot be recovered from, and the program must terminate immediately. The error message is printed to the standard error stream (`stderr`), and the program exits with the provided exit code. After the exit, the program will run the `cleanup` function, which is registered using `atexit`.

3.6.2 Cleanup Function

The `cleanup` function is responsible for deallocating memory and closing any open resources before the program terminates. This is crucial for avoiding memory leaks and resource handling issues. It ensures that any allocated memory is freed and any open files are closed properly.

```
void cleanup() {
    if (lexer) {
        free(lexer);
    }
    if (abstract_syntax_tree) {
        free_node(abstract_syntax_tree);
    }

    if (limits) {
        free(limits);
    }
    if (output_file) {
        fclose(output_file);
    }
}
```

The `cleanup` function is automatically called when the program exits, ensuring that all allocated resources are properly freed. This function frees the lexer, abstract syntax tree, limits, and closes the output file if they were previously opened. This helps maintain the program's efficiency by preventing memory leaks and file handle issues.

3.6.3 Free Node Function

The function `free_node` is responsible for freeing the memory allocated for a node and its child nodes in the Abstract Syntax Tree (AST). It is used during the cleanup process to recursively free all nodes, ensuring that the memory used by the AST is properly deallocated. This is particularly important for preventing memory leaks, especially in recursive data structures such as trees.

```
void free_node(Node *node) {
    if (node == NULL) return;
    if (node->type == NODE_OP) {
        free_node(node->op.left);
        free_node(node->op.right);
    } else if (node->type == NODE_FUNC) {
        free_node(node->func.arg);
    }
    free(node);
}
```

4 User Manual

4.1 Compiling and Building the Program

The program uses a `Makefile` to automate the compilation process. Below are the requirements and steps for different platforms.

4.1.1 Requirements

- `gcc` (for Linux and macOS)
- `MinGW` with `mingw32-make` (for Windows)

4.1.2 Linux and macOS

1. Ensure that `gcc` and `make` are installed.
2. Navigate to the program's directory using the `cd` command.
3. Run the command:
   ```
   make
   ```
4. The resulting executable file will be named `graph`.
5. If the `Makefile` is configured for `graph.exe`, edit it and replace `graph.exe` with `graph`.
6. Run the program with the command:
   ```
   ./graph <function> <output.ps> [<x_min:x_max:y_min:y_max>]
   ```

4.1.3 Windows

1. Install `MinGW` and ensure that `mingw32-make` is available.
2. Navigate to the program's directory using the `cd` command.
3. Run the command:
   ```
   mingw32-make
   ```
4. After compilation, the executable file `graph.exe` will be created.
5. Run the program with the command:
   ```
   graph.exe <function> <output.ps> [<x_min:x_max:y_min:y_max>]
   ```

4.1.4 Viewing .ps Files

- On Linux, `.ps` files can be opened without additional software.
- On macOS and Windows, external software such as `Ghostscript` or `GSview` is required to view `.ps` files.

4.2 Running the Program

The program is executed with the following syntax:

```
graph <function> <output.ps> [<x_min:x_max:y_min:y_max>]
```

Notes:

- The function can be written without quotation marks, but in this case, it must not contain spaces.
- When using quotation marks, the function's notation is more flexible.
- Limits are not necessary. If not provided, will be initialized as default values -10:10:-10:10
- Limits must be logical: `x_min` must be less than `x_max`, and `y_min` must be less than `y_max`.
- Minimum values can be greater than zero.

4.3 Examples of Program Execution

Below are examples of using the program, along with the generated output graphs.

4.3.1 Example 1: Function $\sin(x)$

```
graph "sin(x)" sin_graph.ps -10:10:-2:2
```

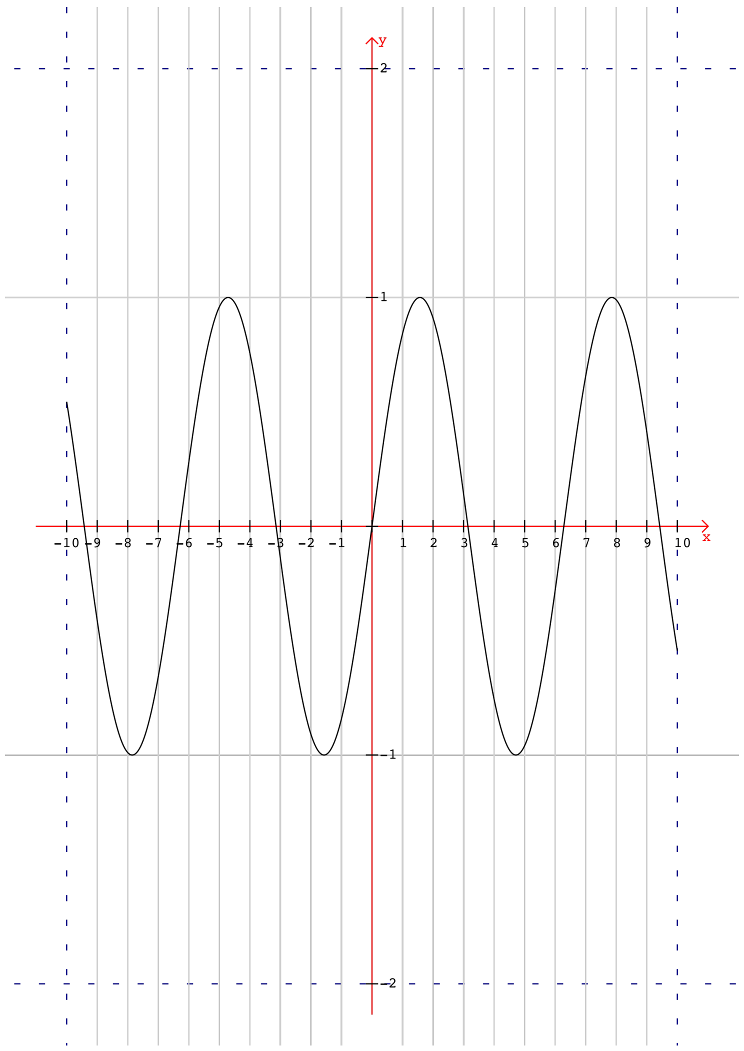


Figure 5: Graph of the function $\sin(x)$ with limits $x \in [-10, 10]$, $y \in [-2, 2]$.

4.3.2 Example 2: Function x^2

```
graph x*x quadratic_graph.ps -5:5:0:25
```

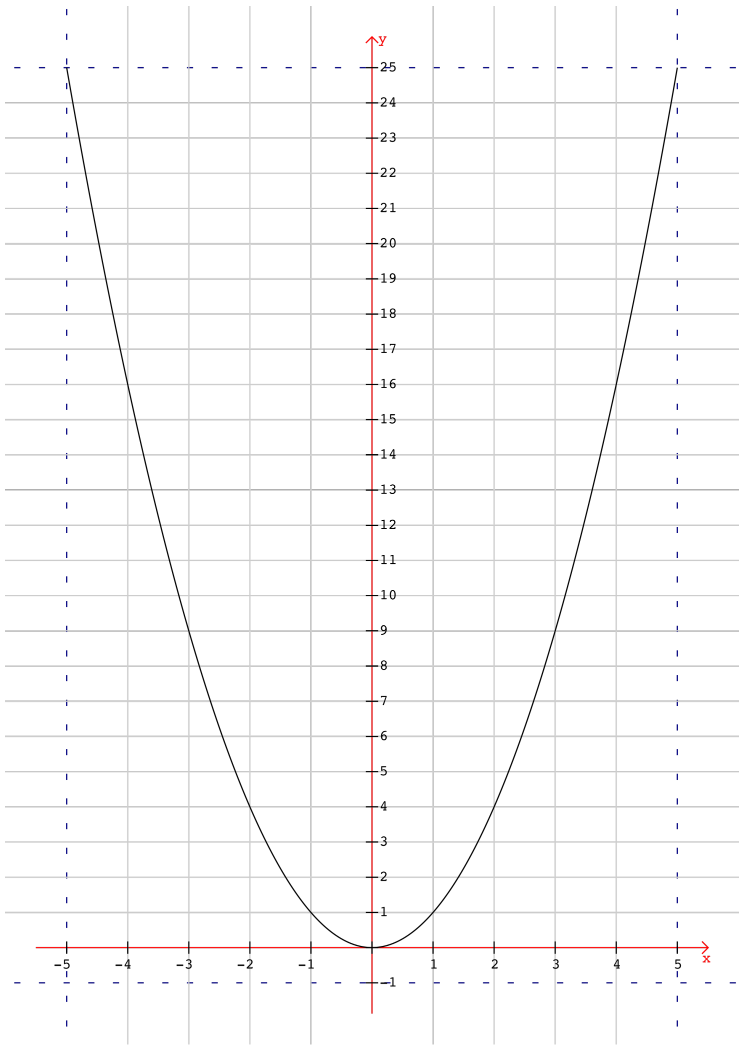


Figure 6: Graph of the function x^2 with limits $x \in [-5, 5]$, $y \in [0, 25]$.

4.3.3 Example 3: Function $\log(x)$

```
graph log(x) log_graph.ps
```

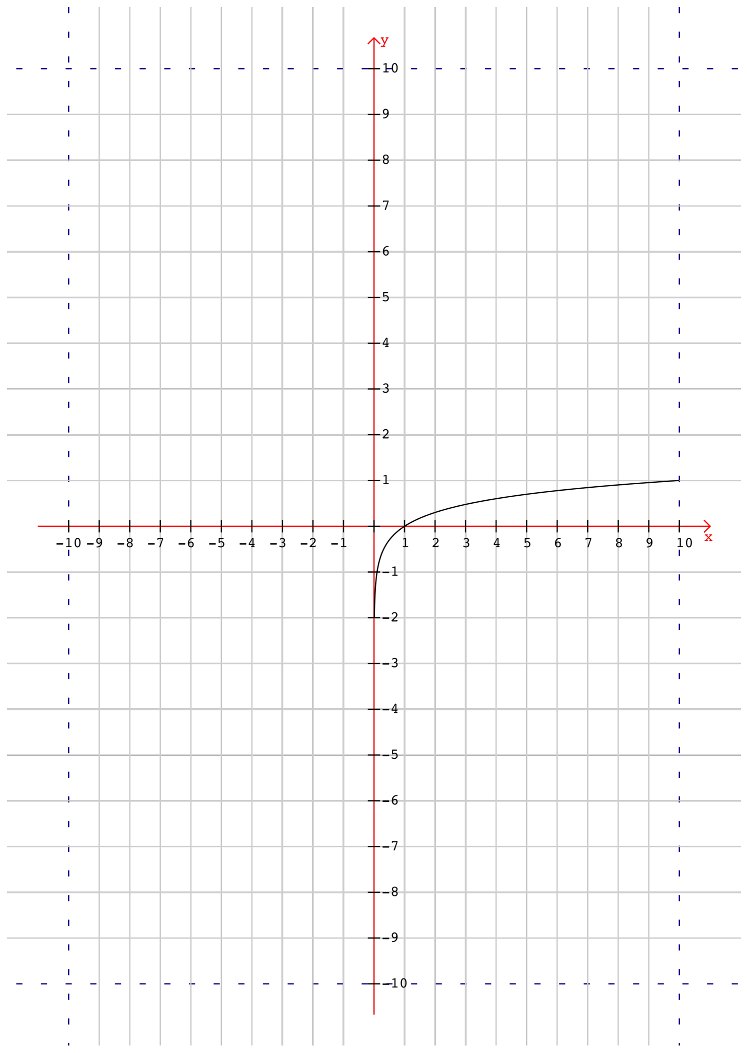


Figure 7: Graph of the function $\log(x)$ with limits $x \in [-10, 10]$, $y \in [-10, -10]$ (Default limits).

5 Conclusion

The program successfully fulfills the project requirements, providing a functional tool for visualizing mathematical functions as graphs in PostScript format. Through the implementation of various modules, the program offers a modular and extendable architecture, ensuring maintainability and clarity of code. Below, we summarize the key results, evaluate the project, and discuss possible improvements and encountered challenges.

5.1 Summary of Achieved Results

- Development of a robust program for graphing mathematical functions.
- Modular design using separate files for core components such as parsing, evaluation, and drawing.
- Comprehensive documentation of all functions, modules, and program usage.
- Execution of the program across multiple platforms (Linux, macOS, Windows) with platform-specific instructions and compatibility.
- Successful generation of accurate function graphs in PostScript format, demonstrated with examples such as $\sin(x)$, x^2, and $\log(x)$.

5.2 Evaluation of Task Fulfillment

The program meets the original task requirements by:

- Supporting function parsing and validation.
- Allowing optional user-defined limits for graphing.
- Generating files compatible with standard PostScript viewers.
- Implementing a command-line interface that is straightforward to use.

The inclusion of detailed documentation further enhances the usability and accessibility of the program.

5.3 Possible and Desirable Improvements

While the program achieves its primary goals, several enhancements could improve its functionality and usability:

- Expanding support for more complex mathematical functions.
- Extending the output format options to include modern image formats such as PNG or SVG for broader compatibility.
- Developing a graphical user interface (GUI) to complement the command-line interface, making the tool more accessible to users unfamiliar with command-line operations.

5.4 Challenges Encountered During Development

Several challenges arose during the project, including:

- Parsing mathematical expressions reliably and constructing an abstract syntax tree for evaluation.
- Generating accurate graphical output and validating the correctness of rendered functions.
- Working on memory leaks, designing project structure.

These challenges were addressed through careful debugging, modular design, and thorough testing.

5.5 Final Remarks

The project demonstrates the application of fundamental programming techniques to solve a practical problem in mathematical visualization. Despite minor limitations, the program provides a solid foundation for further development and can serve as a valuable tool for educational or analytical purposes. The lessons learned during this project, particularly in modular design, will inform future software development endeavors.