Image Manipulation Language

Image Manipulation Domain-Specific Language (DSL) in C

Team Members:

- Saumadeep Sardar — Roll No: CS23B1049
- Anurag Sharma — Roll No: CS23B1062
- Dhage Pratik Bhishmacharya — Roll No: CS23B1047
- JP Akshaya — Roll No: CS23B1074

Course: Compiler Design

Introduction

This project implements a Domain-Specific Language (DSL) in C for image manipulation tasks. The DSL allows users to perform operations such as loading, saving, cropping, blurring, rotating, and applying filters to images using a simple, expressive syntax with features like piping for chaining operations. The system includes a lexer for tokenization, a parser for generating an Abstract Syntax Tree (AST), semantic analysis for type checking, and an evaluator for runtime execution. Built-in functions leverage libraries like STB Image for core image processing. The DSL is interpreted, providing immediate feedback and error handling for invalid operations. Sample scripts demonstrate practical usage, such as thresholding and rotating images.

Image manipulation is a common task in fields like computer vision, graphic design, and media processing. Existing tools (e.g., Photoshop or libraries like OpenCV) are powerful but often complex for simple workflows. A DSL for image manipulation simplifies this by providing concise syntax for operations like filtering, transformations, and compositions. It enables non-experts to chain operations efficiently (e.g., via piping), promotes reusability, and integrates seamlessly with C-based systems. This DSL supports basic to advanced manipulations, making it suitable for educational, prototyping, or lightweight production use.

Lexical Rules and Tokens

The lexer (defined in lexer.l) uses Flex to tokenize input scripts. It recognizes keywords, literals, operators, and identifiers while ignoring whitespace.

Key Lexical Rules:

- **Identifiers (ID):** [a-zA-Z_][a-zA-Z0-9_]* - Variable names or function names (e.g., img, load).
- **Integer Literals (INT_LIT):** "-"?[0-9]+ - e.g., 10, -5.
- **Float Literals (FLOAT_LIT):** [0-9]+"."[0-9]*([eE][+-]?[0-9]+)? - e.g., 3.14, 1e-2.
- **String Literals (STR_LIT):** "\"([^\\\"]|\\.)*\"" - e.g., "input.jpg". Supports escapes like \n, \t.
- **Operators:** |> (PIPE_OP), = (ASSIGN), == (EQ), != (NEQ), > (GT), < (LT), >= (GE), <= (LE), + (PLUS), - (MINUS), * (MUL), / (DIV), % (MOD).
- **Keywords:** def (DEF), return (RETURN), if (IF), else (ELSE), for (FOR), while (WHILE), break (BREAK), continue (CONTINUE), true (TRUE), false (FALSE), null (NULLVAL).
- **Type Keywords:** int (INT_TK), float (FLOAT_TK), string (STRING_TK), image (IMAGE_TK), bool (BOOL_TYPE).
- **Built-in Functions (Treated as IDENT):** load, save, crop, resize, scale, rotate, flipX, flipY, blur, sharpen, grayscale, invert, brighten, contrast, threshold, cannyedge, blend, mask, print.
- **Whitespace:** [\t\n\r]+ - Ignored.

Unknown characters trigger an error and exit.

Grammar and Parser Description

The parser (in parser.y) uses Bison to generate an SLR(1) parser. It builds an Abstract Syntax Tree (AST) from tokens, supporting statements like declarations, assignments, control flow, and expressions with piping.

Key Grammar Rules:

- **Program:** program: stmt_list - Root is a block of statements.
- **Statements (stmt):** Includes declarations (e.g., int a = 10;), assignments (e.g., img = load("input.jpg");), expressions, returns, if/else, while, for, break, continue, function definitions.
- **Declarations:** declaration: type IDENT ASSIGN expr ';' - e.g., image img = load("input.jpg");.
- **Expressions (expr):** Supports primaries and piping: expr: primary_expr | expr PIPE_OP primary_expr.
- **Primary Expressions (primary_expr):** Literals (int/float/string), identifiers, function calls.
- **Function Calls (call):** IDENT '(' expr_list_opt ')' - e.g., load("input.jpg").
- **Blocks:** '{' stmt_list '}' or single statements.
- **Parameter Lists:** For function definitions, e.g., def func(x, y) { ... }.
- **Precedence:** PIPE_OP is left-associative; expects 1 shift/reduce conflict (for commas).

The parser generates an AST using constructors from ast.c (e.g., make_decl_node, make_pipe).

Semantic Rules

Semantic analysis occurs during evaluation in eval.c, enforcing type safety and runtime checks.

- **Type Propagation:** AST nodes carry types (e.g., TypeId enum: TYPE_INT, TYPE_FLOAT, TYPE_STRING, TYPE_IMAGE).
- **Declarations:** Check initializer type matches declared type (e.g., cannot assign string to int). Coercions: int to float, float to int (with truncation).
- **Assignments:** Overwrite variables, but type mismatches cause runtime errors.
- **Function Calls:** Built-ins validate argument counts and types (e.g., load expects 1 string; crop expects image + 4 ints).
- **Piping:** LHS becomes first argument of RHS call; types must align (e.g., image output piped to image-input function).
- **Value Handling:** Uses Value union for tagged types; clones images/strings to avoid ownership issues.
- **Environment:** Global symbol table (env_set, env_get) stores variables; frees resources on shutdown.

Errors are raised via runtime_error for mismatches, undefined variables, or invalid operations.

Syntax

File I/O

Load an image

- Loads an image from a file into a variable.
- **Example:** `img1 = load("input.png")`

Save an image

- Saves an image variable to a file.
- **Syntax:** `save(filename_string, image_variable)`
- **Example:** `save("output.jpg", img1)`

Geometric Transforms

Crop an image

- Extracts a rectangular region from an image.
- **Example:** `img_cropped = img |> crop(10, 10, 100, 100)`

Resize an image

- Resizes an image to specific dimensions using nearest-neighbor scaling.
- **Example:** `img_resized = img |> resize(800, 600)`

Scale an image

- Scales an image by a floating-point factor.
- **Example:** `img_scaled = img |> scale(2.5)`

Rotate an image

- Rotates an image by 90 degrees.
- **Example:** `img_rotated = img |> rotate(1)` (Note: `direction` 1 for CW, -1 for ACW)

Flip Horizontal (X-axis)

- Flips an image horizontally.
- **Example:** `img_flipped = img |> flipX()`

Flip Vertical (Y-axis)

- Flips an image vertically.
- **Example:** `img_flipped = img |> flipY()`

Filters & Effects

Blur an image

- Applies a blur filter.
- **Syntax:** `new_img = blur(radius_int)`
- **Example:** `img_blurred = img |> blur(5)`

Sharpen an image

- Applies a sharpening filter.

- **Syntax:** `new_img = sharpen(amount_int, direction_int)`
- **Example:** `img_sharp = img |> sharpen(10, 1)` (Note: `direction` 0 for reducing, 1 for increasing)

Canny Edge Detection

- Runs the Canny edge detection algorithm.
- **Syntax:** `new_img = cannyedge(sigma_float, low_threshold_int, high_threshold_int)`
- **Example:** `img_edges = img |> cannyedge(1.4, 50, 100)`

Color & Tone Adjustments

Grayscale

- Converts an image to grayscale.
- **Example:** `img_gray = img |> grayscale()`

Invert Colors

- Inverts the colors of an image.
- **Example:** `img_inverted = img |> invert()`

Adjust Brightness

- Increases or decreases image brightness.
- **Syntax:** `brighten(bias_int, direction_int)`
- **Example:** `img_brighter = img |> brighten(20, 1)` (Note: `direction` 0 for reducing, 1 for increasing)

Adjust Contrast

- Increases or decreases image contrast.
- **Syntax:** `contrast(amount_int, direction_int)`
- **Example:** `img_contrast = img |> contrast(30, 1)` (Note: `direction` 0 for reducing, 1 for increasing)

Apply Threshold

- Applies a binary threshold to the image.
- **Syntax:** `threshold(threshold_int, direction_int)`

- **Example:** `img_thresh = img |> threshold(128, 1)` (Note: `direction` 0 for lower is white, 1 for upper is white)

Compositing

Blend two images

- Blends two images together using an alpha value.
- **Syntax:** `blend(image2, alpha_float)`
- **Example:** `img_blended = img |> blend(img2, 0.5)`

Mask an image

- Applies one image as a mask to another.
- **Syntax:** `mask(mask_image)`
- **Example:** `img_masked = img |> mask(img_mask)`

Utility

Print Image Info

- (Inferred) Prints information about an image variable, such as its dimensions.
- **Syntax:** `print(image_variable,int,float,string, . . .)`
- **Example:** `print(img1)`

Code Generation & Execution

The system is interpreted, not compiled to machine code. After parsing, the AST is evaluated directly.

- **AST Construction:** Parser builds AST nodes (e.g., Ast struct with unions for variants).
- **Evaluation (eval_program):** Traverses the AST:
    - Statements: Execute declarations, assignments, control flow.
    - Expressions: Evaluate literals, identifiers, calls, pipelines.

    - Built-ins: Dispatched via eval_builtin_call, calling runtime functions (e.g., load_image from runtime.c).
- **Runtime Integration:** Uses stb_image for I/O, custom implementations for filters (e.g., box blur, Canny edge).
- **Execution Flow:** yyparse() builds AST; eval_program(root) runs it; frees AST and environment.

No intermediate code generation; direct interpretation for simplicity.

Detailed Code Walkthrough

This section provides a step-by-step explanation of each key file in the project. We analyze the code structure, key functions, and logic flow, highlighting how components interact. Code snippets are annotated with inline explanations for clarity.

1. ast.h - Abstract Syntax Tree Definitions

This header defines the AST structure, enums, and constructor prototypes. It serves as the core data model for parsed code.

1. **Enums Definition:**
    - TypeId: Enumerates types (TYPE_INT, TYPE_FLOAT, TYPE_STRING, TYPE_IMAGE, TYPE_UNKNOWN). Used for type annotations in nodes.
    - AstType: Lists all node variants (e.g., AST_INT_LIT for integer literals, AST_DECL for declarations, AST_PIPELINE for piping).
2. **Ast Struct:**
    - Contains type (AstType), type2 (TypeId for propagation), and a union for variant-specific data.
    - Examples:
        - decl: { Ast *type_node; char *name; Ast *expr; } - For typed declarations like int a = 10;.
        - pipe: { Ast *left; Ast *right; } - For pipelines like load("img.jpg") |> blur(5).
        - call: { char *name; Ast **args; int nargs; } - For function calls.
3. **Constructors:**
    - Prototypes like make_int_literal(int v): Allocates and initializes an Ast node with type = AST_INT_LIT, ival = v.
    - make_decl_node(Ast *type_node, char *name, Ast *expr): Links type, name, and initializer.
    - Utility: free_ast(Ast *ast) - Recursively frees tree; dump_ast(Ast *ast, int indent) - Pretty-prints for debugging.

2. ast.c - AST Constructors and Utilities

Implements the constructors and utilities declared in ast.h. Handles memory allocation and node initialization.

1. **Literal Constructors:**
    - make_int_literal(int v):
        - Step 1: Ast *ast = malloc(sizeof(Ast)); - Allocate node.
        - Step 2: Set ast->type = AST_INT_LIT; ast->type2 = TYPE_INT; ast->ival = v;.
        - Returns ast or NULL on failure.
    - Similar for make_float_literal(double v) (sets fval) and make_string_literal(const char *s) (duplicates s into sval).
2. **Type and Declaration Nodes:**
    - make_type_node(TypeId t): Allocates AST_TYPE node with type2 = t.
    - make_decl_node(Ast *type_node, char *name, Ast *expr):
        - Allocates AST_DECL node.
        - Propagates type: ast->type2 = type_node->type2;.
        - Duplicates name; links type_node and expr.
3. **Existing Constructors (e.g., Assignments, Calls):**
    - make_assign(char *name, Ast *expr): Sets AST_ASSIGN, duplicates name.
    - make_call(char *name, Ast **args, int nargs): Duplicates name, assigns args array.
    - make_pipe(Ast *left, Ast *right): Links left/right for chaining.
4. **Block and Control Flow:**
    - make_block(Ast **stmts, int n): Assigns statement array and count.
    - Loops like make_if, make_while: Link condition and body.
5. **Free and Dump Utilities:**
    - free_ast(Ast *ast): Switch on type; recursively free children (e.g., for AST_DECL: free_ast(ast->decl.type_node); free(ast->decl.name); free_ast(ast->decl.expr);). Frees strings/arrays; ignores primitives.
    - dump_ast(Ast *ast, int indent): Indent printing; switch cases print node details (e.g., "Int: %d" for literals, recursive dumps for composites).

3. eval.h - Evaluator Interface

Defines the Value type, environment functions, and evaluator prototypes. Bridges AST to runtime.

1. **Value Type:**
    - ValueType enum: V_INT, V_FLOAT, V_STRING, V_IMAGE, V_NONE.
    - Value struct: tag (type) + union { int ival; double fval; char *sval; Image *img; } u;.
2. **Prototypes:**
    - eval_program(Ast *prog): Entry point; evaluates top-level block.
    - eval_stmt(Ast *stmt): Handles statements.
    - eval_expr(Ast *expr): Returns Value; changed from void for expression results.

- Environment: env_set(const char *name, Value val), env_get(const char *name) - Global vars.
- Utils: runtime_error(const char *format, ...) - Fatal errors; free_value(Value val) - Frees heap data (strings/images); env_shutdown() - Cleans globals.

4. eval.c - Core Evaluator

Implements AST traversal, type coercion, built-in dispatching, and environment management.

1. **Symbol Table:**
   - Var linked list: { char *name; Value val; Var *next; }; static Var *globals = NULL;.
   - env_set: Search for existing; free old val, assign new. If new: allocate, duplicate name, prepend.
   - env_get: Linear search; error if missing.
2. **Error and Value Helpers:**
   - runtime_error: vfprintf to stderr, exit(1).
   - val_none(): V_NONE value.
   - free_value: Frees sval (free) or img (free_image).
   - Coercions: value_to_int (float truncates), value_to_float (int promotes), etc. - Error on mismatch.
   - copy_image: Deep clone Image (memcpy data); errors on null/invalid.
   - value_clone: Duplicates strings/images; copies primitives.
3. **Built-in Dispatcher (eval_builtin_call):**
   - Consumes args (frees them).
   - Examples:
     - load: 1 string arg; calls load_image, sets V_IMAGE.
     - save: 2 args (string path, image); calls save_image.
     - threshold: Image + int threshold + int direction; calls apply_threshold.
     - rotate: Image + int direction (±1 for 90°); calls rotate_image_90.
     - print: Variable args; handles strings/images (escaped print).
   - Defaults to V_NONE.
4. **Statement Evaluation (eval_stmt):**
   - Switch on type:
     - AST_DECL: Eval expr to Value val; coerce/check against type2 (e.g., int/float); env_set.
     - AST_ASSIGN: Eval expr, overwrite in env (no type check yet).
     - AST_EXPR_STMT: Eval and free result.
     - Others: Stubbed (e.g., AST_FUNC_DEF - TODO).
5. **Expression Evaluation (eval_expr):**
   - Literals: Return new Value (duplicate strings).
   - AST_IDENT: Clone from env_get.
   - AST_CALL: Eval args to array; dispatch to built-in; free args.
   - AST_PIPELINE: Eval left (lhs); right must be call; prepend lhs to args; dispatch.
   - Obsoletes: Error on old nodes.

6. **Program Entry (eval_program):**
    - Loop over block stmts; call eval_stmt.
    - Call env_shutdown: Traverse globals, free names/values/Var.

5. runtime.h - Image Runtime Prototypes

Declares Image struct and operations.

- Image: { int width, height, channels; unsigned char *data; } - RGB-focused (channels=3).
- Functions: load_image, save_image, crop_image (x,y,w,h), blur_image (radius), grayscale_image, invert_image, flip_image_along_X/Y, run_canny (sigma, thresholds), adjust_brightness/contrast (bias/amount, direction), apply_threshold (threshold, direction), convolve_image (3x3 kernel), sharpen_image (amount, direction), blend_images (img1, img2, alpha), mask_image (img, mask), resize_image_nearest (new_w, new_h), scale_image_factor (factor), rotate_image_90 (direction ±1), print_string_escaped (handles \n, \t).

6. runtime.c - Image Processing Implementations

Uses STB for I/O; custom algos for filters. (Truncated, but key parts explained.)

1. **Load/Save:**
    - load_image: stbi_load with 3 channels (RGB); allocate Image.
    - save_image: stbi_write_png with 3 channels.
2. **Crop (crop_image):**
    - Validate params/bounds.
    - Allocate output Image (w x h x 3).
    - Loop over rows: memcpy row slices from source offsets.
3. **Blur (blur_image):**
    - Box filter: For each pixel, sum neighbors in radius; average per channel.
    - Uses long long sum[3] to avoid overflow; clamps to unsigned char.
4. **Free (free_image):** stbi_image_free(data); free(img);.
5. **Grayscale (Partial):**
    - Allocate output; convert RGB to gray (standard formula implied).
6. **Resize (resize_image_nearest):**
    - Ratios: x_ratio = old_w / new_w.
    - Nearest neighbor: Map dst (x,y) to src floor; memcpy pixels.
7. **Scale (scale_image_factor):** Calls resize with scaled dims.
8. **Rotate 90° (rotate_image_90):**
    - Swap w/h.
    - For clockwise (1): x_src = y_out; y_src = h_in - 1 - x_out.
    - Copy RGB channels pixel-by-pixel.
9. **Print Escaped:** Loop chars; handle \, \n, \t, etc.

7. parser.y - Bison Grammar

Defines grammar rules; actions build AST.

1. **Tokens/Types:** %token for keywords/operators; %union for values; %type <ast> for rules.
2. **Start:** program: stmt_list { root = $1; }.
3. **Type Rule:** type: INT_TK { $$ = make_type_node(TYPE_INT); } etc.
4. **Declaration:** type IDENT ASSIGN expr ';' { $$ = make_decl_node($1, $2, $4); } - Note: Fixed from = to ASSIGN.
5. **Stmt List:** Recursive: Single stmt → block(1); append to existing.
6. **Stmts:** Dispatch to constructors (e.g., IF '(' expr ')' block { $$ = make_if($3, $5); }).
7. **Expr:** expr PIPE_OP primary_expr { $$ = make_pipe($1, $3); } - Chains left-assoc.
8. **Primary/Call:** Literals to constructors; calls build arg arrays via expr_list (realloc on append).
9. **Params:** params_list: IDENT | params_list ',' IDENT { $$ = append_arg($1, $3); } - Builds char** list.

8. lexer.l - Flex Lexer

Tokenizes input; sets yylval.

1. **Patterns:** Keywords return tokens (e.g., "def" { return DEF; }).
2. **Built-ins:** As IDENT with strdup(yytext) (e.g., "load").
3. **Literals:** INT_LIT via atoi; FLOAT_LIT via atof; STR_LIT strips quotes, strdup.
4. **Operators/Punct:** Return tokens.
5. **ID:** Fallback {ID} { yylval.str = strdup(yytext); return IDENT; }.
6. **Ignore:** Whitespace; error on unknowns.

9. script.iml - Sample Script

Demonstrates DSL usage.

1. int a= 10;: Declares int var (unused).
2. img = load("input3.jpg") |> threshold(180,1);: Load image; pipe to threshold (img first arg).
3. img2 = load("input2.jpg") |> rotate(-1);: Load, pipe to counterclockwise 90° rotate.
4. save("output.jpg",img2);: Save rotated image.
5. print("done running", img,"\n");: Print string + image ref + newline (escaped).

Code Flow Analysis

1. **Input Script:** Read from file (e.g., script.iml).
2. **Lexing:** yylex() tokenizes input, passing to parser.
3. **Parsing:** yyparse() constructs AST, storing in root.
4. **Evaluation:**
    - eval_program: Loops over block statements.
    - For declarations/assignments: Evaluate RHS, check types, store in env.
    - For calls: Evaluate args, dispatch to built-ins.
    - For pipes: Evaluate left, prepend to right's args.
5. **Runtime Operations:** Built-ins manipulate Image structs (e.g., allocate/copy data).
6. **Output:** Saves images or prints via built-ins.
7. **Cleanup:** free_ast, env_shutdown (frees images/strings), free_image.

Flow is sequential, with control structures altering it (e.g., loops, conditionals).

Error Handling

- **Lexical/Parse Errors:** Flex/Bison report unknowns or syntax issues (e.g., via yyerror).
- **Semantic/Runtime Errors:** runtime_error exits with messages for:
    - Type mismatches (e.g., "Type error: expected image, got 0").
    - Argument count errors (e.g., "load() expects 1 argument, got 2").
    - Undefined variables ("Variable 'x' not found").
    - Invalid operations (e.g., null images, out-of-bounds crop).
    - Memory failures.
- **Image-Specific:** Checks for null data, invalid dimensions in runtime functions.

No recovery; errors terminate execution.

Sample Input/Output

Contributions

- ➔ Saumadeep Sardar (CS23B1049):
  - ◆ Lexer, parser, AST implementation
- ➔ Anurag Sharma (CS23B1062):
  - ◆ Image manipulation algorithms implementations.
  - ◆ Handling Image Memory Management to avoid segmentation Faults
- ➔ Dhage Pratik Bhishmacharya (CS23B1047):
  - ◆ Arithmetic, relational evaluations. Conditionals and loops implementation.
- ➔ JP Akshaya (CS23B1074):
  - ◆ Intermediate code generation from the evaluation which directly executed code.

References

- Flex & Bison Documentation: https://www.oreilly.com/library/view/flex-bison/9780596805418/
- STB Image Library: https://github.com/nothings/stb
- Canny Edge Detection: Adapted from public implementations and GenAI.
- Online Resources: Stack Overflow for image algorithms, GNU Bison manual.