IMAGE MANIPULATION LANGUAGE

DSL for Image Manipulation In C

Made using **Four Brain Cells**

TEAM MEMBERS

CS23B1049 Saumadeep Sardar	Lexer, Parser and AST(Abstract syntax tree) base code
CS23B1062 Anurag Sharma	Image manipulation and Image memory handling. Function call evaluations.
CS23B1074 JP Akshaya	Intermediate code generation from the evaluation which directly executed code.
CS23B1047 Dhage Pratik B	Arithmetic, relational evaluations. Conditionals and loops implementation.

INTRODUCTION

- This project implements a Domain-Specific Language (DSL) in C aka Image Manipulation Language (IML) for image manipulation tasks.
- The IML 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.
- It can be used both as an interpreted and compiled way whatever is suitable.

MOTIVATION - WHY AN IMAGE MANIPULATION LANGUAGE?

- Image manipulation is fundamental in multiple domains from **Robotics, AI preprocessing** and **computer vision**.
- Existing powerful tools (e.g., OpenCV, complex libraries) are **overkill** for simple, repetitive image workflows.
- Just loading, cropping, and saving an image requires many lines of boilerplate code in Python.

PROJECT OVERVIEW

The project contains the following files:-

- **main.c** - Entry point: reads script file, calls yyparse(), starts AST evaluation
- **lexer.l** - Converts raw script text into **tokens** (IDENT, STR_LIT, PIPE_OP, etc.)
- **parser.y** - Parses tokens using grammar rules → builds **AST** by calling node constructors
- **ast.h / ast.c** - Defines & creates **AST nodes** (make_call, make_pipe, make_assign, etc.)
- **eval.h / eval.c** - Interprets AST: evaluates expressions, manages variables, calls built-ins
- **runtime.h / runtime.c** - Executes image operations: load_image(), blur(), rotate(), save(), etc.
- **compile.c -** generates the C code for the language and creates an executable file(a.out) to run the program.
- **script.iml** - Example input script using DSL syntax with piping (|>)
- **include/** - includes the image manipulation functions to load and write images in C.

The IML can directly run scripts and also generate an executable to be used as when needed.

LEXER & TOKENIZATION

Lexer Implementation

- Tool: Flex (lexer.l) - Scans input to tokens
- Handles: Keywords (def, image), literals (int/float/str)
- Operators: |> (PIPE_OP) = (ASSIGN)

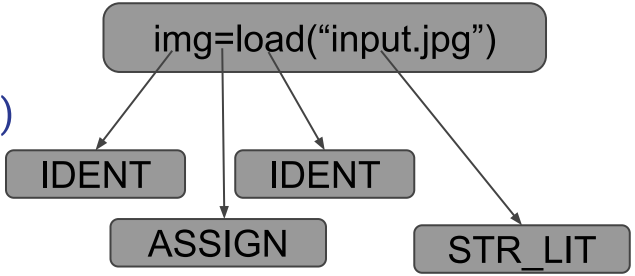


Fig: Tokenization

Key Rules

- Built-ins as IDs: e.g., "load" → IDENT with strdup
- Strings: ("file.jpg") → STR_LIT

Example Output

- Input: "img = load("input.jpg")"
- Tokens: IDENT | ASSIGN | IDENT | STR_LIT

PARSER & AST

Parser Implementation

- Tool: Bison (parser.y) - Builds AST from tokens (LALR(1))
- Rules: Stmts (declarations/calls), pipes (left-associative)
- Dynamic: Realloc for arg lists in calls

AST Structure

- Nodes: Union (AST_DECL, AST_PIPE, AST_CALL)
- Constructors: e.g., make_pipe(left, right) from ast.c

Example

- load("input.jpg") |> blur(5) → Pipe node with call children

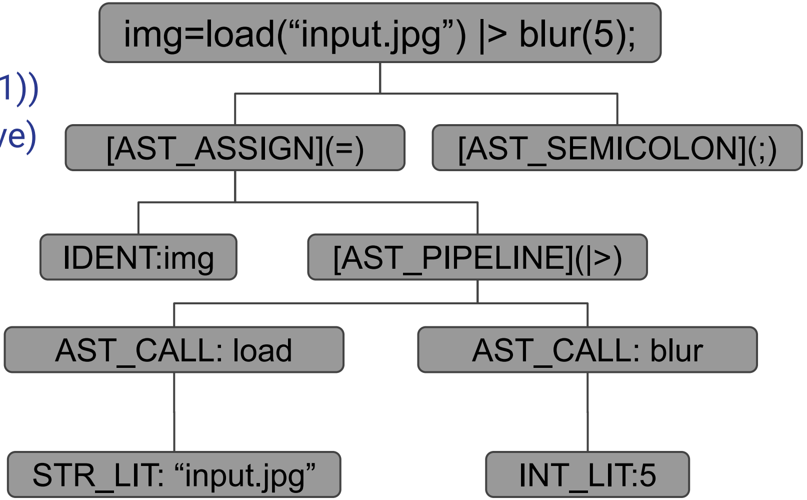


Fig: Parse tree

THE CORE ARITHMETIC AND CONTROL FUNCTIONALITIES

- Implementation of the loops and conditional statements.
- Resolving and evaluating the arithmetic and relational operations.
- Evaluating declarations and expressions properly.

Semantics

- Types: Enum (INT/FLOAT/STRING/IMAGE); Coercions (int→float)
- Checks: Runtime errors for mismatches

IMAGE MANIPULATION BUILT-IN FUNCTIONS

```
"load"
"save" "crop"
"resize" "scale"
"rotate""flipX"
"flipY" "blur"
"sharpen" "grayscale"
"invert" "brighten"
"contrast" "threshold"
"cannyedge" "blend"
"mask" "print"
```

Implemented these built-in functions and the evaluation for their calls respectively.

EVALUATOR

Evaluator Core

- eval.c: Recursive AST traversal (eval_program → eval_stmt → eval_expr)

Piping Handling

- LHS value prepends to RHS args; Dispatches to built-ins

RUNTIME IMPLEMENTATION

Runtime Layer

- runtime.h/c: Image struct (width/height/channels/data)
- I/O: STB for load/save (force RGB=3 channels)

Key Functions

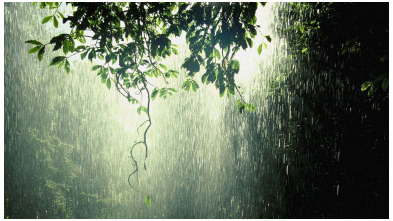
- Crop: Memcpy row slices with bounds check
- Blur: Box filter (sum neighbors → avg per channel)
- Rotate: 90° coordinates remap + pixel copy

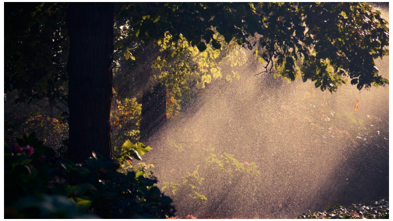
and more…

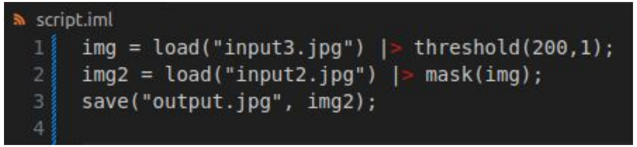
INTEGRATION & RESULTS

Full Pipeline

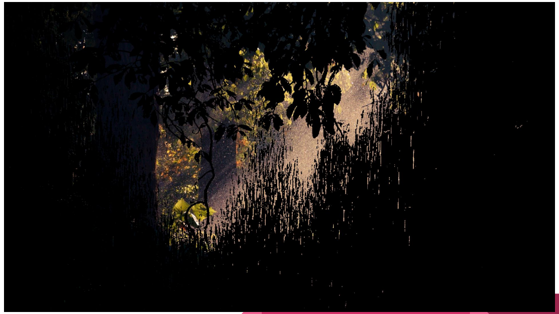
- Flow: yyparse (AST) → eval_program → Built-in calls → Cleanup







```
img = load("input3.jpg") |> threshold(200,1);
img2 = load("input2.jpg") |> mask(img);
save("output.jpg", img2);
```



COMPILER BACK END

- Converted the structure from an **Interpreter** (executing the AST via `eval.c`) to a **Compiler** (generating source code via `compile.c`).
- Implemented AST traversal functions that use fprintf to generate portable C source code (generated_code.c), effectively translating the high-level DSL syntax into C.

Output pipeline

- Code Generation (compile.c) —> generated_code.c.
- Target Compilation (gcc) —> a.out.
- Execution (./a.out) —> Final Image.

CHALLENGES FACED

Memory Management Issues

- Leaks in AST nodes & image clones caused segmentation fault when accessing from variable names. Solved by value_clone() by creating deep copy for data.

Escape Sequence Implementation

- Needed to manually process strings for escape sequences

Null Implementation

- Implementation of null needed a lot a changes in the handling with relational operators.

RUNTIME ISSUES

Image Data Handling

- Nulls/bounds in crop/resize → Crashes
- Validate params + fprintf errors before ops

STB Library Quirks

- Channel mismatches (grayscale vs. RGB)
- Force 3 channels in load/save

Piping Edge Cases

- RHS not a call → Invalid
- So we put strict type checks in eval_expr

Thank You !