Pixel

Pixel Final Report

Alex Anthony Cortes-Ose (ac4441)
Dillon Davis (dhd2121)
Jessica Kim (sk4711)
Jessica Peng (jp3864)

April 26, 2021

Table of Contents

1 Introduction

Pixel is a language that is designed to process and manipulate images. The idea for our language was originally inspired by the concept of applying filters to images. On social media platforms such as Snapchat and Instagram, there are pre-set filters that people are able to apply to their photos. Every user has access to these filters, and applying these filters may alter each different photo in its own unique way depending on the state of the original photo.

We wanted to create a language that could not only create these filters, but perform the various tasks which are fundamental to basic image processing. With our language, it is easy to write algorithms that perform enhancement, transformation, edge detection, and basic analysis on images (see section 2, use cases). An *image* is its own primitive type, containing a *pixel* matrix; *pixel* is another primitive type. These types enable extended functionality and make available information useful in many image processing applications.

The syntax for our language draws from Python and JavaScript in several notable ways. The design of Python libraries and its array and matrix manipulation features are crucial for image processing, and we admire the simplicity of Python in this aspect. We were also inspired by the map, forEach, and reduce (which is equivalent to fold_left) methods for lists in JavaScript, the explicit code structure with semicolons and easy scoping with brackets, the language's arrow syntax for anonymous function declaration, and its support for higher order functions.

2 Use Cases

Our language can be used to write and evaluate many fundamental image processing algorithms and can achieve various image enhancements, transformations, and analyses. Some of these include:

- Noise removal
- Image sharpening
- Thresholding
- Color shifting
- Blurring
- Edge and feature detection (texture analysis corners)
- Blob and object detection
- Contrast enhancement
- Rotation
- Enhanced mapping
- Altering RGB order
- Flipping pixel order (flip image)
- Convert image to black and white
- Convert image to grayscale
- Encrypt image
- Rank filters, thresholding
- Create an RGB representation of a gray-level image

3 Language Details

3.1 Data Types

Data Type	Description	
`int`	4-byte signed integer type	`int integer = 4;`
`str`	Array of ASCII characters	`str string = “hello world”;`
`float`	8-byte floating point number	`float fl = 3.5;`
`matrix`	Mutable data structure storing 2 dimensions of floats	`matrix::float mat = [[1, 0, 0, 1],`

		`[0, 1, 0, 1],` `[0, 0, 0, 0]];`
`image`	A specific type of matrix that represents an image consisting of each row as a list of pixels within an outer list	

In our language, a boolean true or false value is represented by the integer values of either 1 or 0, respectively. This enables thresholding and binary image generation, a common image manipulation task.

3.2 Keywords

The following keywords are reserved for this language:

`fun, matrix, image, int, float, while, for, if, else, return, str, void`

3.3 Operators

Operator	Operation
`+`	Scalar, matrix-scalar, element-wise matrix addition
`*`	Scalar, matrix-scalar, matrix (dot product) multiplication
`>, >=, <, <=, ==, !=`	Scalar comparison
`**`	Element-wise matrix exponentiation

Any combination of these basic operators when used with scaling can enable behaviors such as subtraction, division. Loops enable element-wise multiplication and division when paired with assign - these are not very commonly used. The dot product between two matrices can be done with *, and comparison operators return either 1 for true or 0 for false.

3.3 Built-in Matrix and Image Functions

Function	Description	Return type

`print()`	Prints any given data type to stdout	`void`
`image_in(str filePath, str mode)`	Converts either a filepath string into an image type according to mode - mode = “COLOR” enables red, green, blue channels, mode = “GRAYSCALE” enables only the grayscale channel.	`image`
`image_out(str filePath, image i, str mode)`	Saves an image onto the user’s hard drive at location specified in the file path. i is the image type being saved.	`image`

`join(matrix R, matrix G, matrix B)`
OR
`join(matrix GS)`
	Combines either red, green, and blue channels into an image type, or a single grayscale channel into an image type. This enables a user to save the changes to a matrix as an image.	`image`

`convolute(matrix src, matrix kernel)`	Convolutes the src matrix with the kernel matrix, returning a new convoluted matrix	`matrix`

3.4 Built-in Image Functions

3.5 Matrix Functionality

Name	Description	Return Type
`zeros(int r, int c)`	Creates a matrix of zeros with r rows and c columns	`matrix`

Name	Description
`matrix``.rows`	An int describing the number of rows of a given matrix
`matrix``.cols`	An int describing the number of columns of a given matrix

3.6 Features

- Single line comment: `//`
- Multi line (nestable) comment: `/* */`
- Array and matrix indexing is done in the style of numpy array slicing,

```
matrix::float m = [
        [10, 10, 10],
        [10, 10, 10],
        [10, 10, 10]
   ];

float first_row = m[0];
```

4 Code Samples

1. Apply a 1/9 box blur to a cat image:

```
fun::void main() {
  image base = image_in("../img/cat.png");
  matrix::float box =
    [[1, 1, 1],
     [1, 1, 1],
     [1, 1, 1]];
  matrix::float fr = convolute(base.red, box) * 0.11;
  matrix::float fg = convolute(base.green, box) * 0.11;
  matrix::float fb = convolute(base.blue, box) * 0.11;

  image final = join(fr, fg, fb);
  image_out("../img/blurred-cat.png", final);
}
```

2. Threshold an image of flowers to find centers:

```
fun::void main() {
  image base = image_in("../img/flowers.JPG");
  matrix::float base_gray = base.grayscale;
```

```
  matrix::float final = zeros(base_gray.rows, base_gray.cols);
  int i;
  for (i = 0; i < base_gray.rows; i = i + 1) {
    int j;
    for (j = 0; j < base_gray.cols; j = j + 1) {
      if (base_gray[i,j] >= 0.58) {
        final[i, j] = 1;
      }
    }
  }

  image_out("../img/flower_centers.JPG", final);
}
```

3. Apply sobel convolutions to find edges of a steam engine:

```
fun::void main() {
  image base = image_in("../img/steam-engine.png");
  matrix::float base_gray = base.grayscale;
  matrix::float sobel_horizontal =
    [[1, 0, -1],
     [2, 0, -2],
     [1, 0, -1]];
  matrix::float horiz = convolute(base_gray, sobel_horizontal);

  matrix::float sobel_vertical =
    [[1, 2, 1],
     [0, 0, 0],
   [-1, -2, -1]];
  matrix::float vert = convolute(base_gray, sobel_vertical);

  matrix::float fhoriz = horiz * horiz;
  matrix::float fvert  = vert  * vert;
  matrix::float final = (fhoriz + fvert) ** (1/2);

  image_out("../img/steam-engine-edges.png", final);
}
```

5 Project Plan

5.1 Development Process

- We made sure to meet as regularly as our schedules allowed – oftentimes we were busy but made late nights after PLT class work to discuss plans, task distribution, and work. We wanted to prioritize the Parser, Scanner, and AST after the Hello World milestone so that we could access files in our own language, and then made sure that a full-fledged utils.cpp OpenCV suite of C++ functions worked to apply any image filters and convolutions we wanted. This left us to link the utils.o file and generate code for the new semantically checked language in accordance with a refined SAST.

5.2 Software Development Tools

- Libraries and languages: OCaml version 4.05.0, including Ocamlyacc and Ocammllex and LLVM. OpenCV 4.5.1_2 installed in Mac homebrew.

5.3 Style Guide

- We made sure to keep indentations at 2 spaces, and named variables according to Python convention (except for the util.cpp where we used camelCase).

5.4 Team Roles

Member	**Role**
Alex Anthony Cortes-Ose	Language Guru
Dillon Davis	Manager
Jessica Kim	Tester

Jessica Peng	System Architect

6 Test Suites

We created around 60 test suites, in order to test the syntax, grammar and functionality of Pixel. Some test that a code sample produces an expected error, while others test for an expected output, such as the printing of specific integers or strings. Below are some examples of our test suites, along with brief descriptions of each.

- fail-nomain.pixel – tests that a program with no main function will produce an error
- fail-return1.pixel – tests that a program returning the wrong type will produce an error
- fail-while1.pixel – tests that using the wrong type of predicate will produce an error
- test-add1.pixel – tests that integer addition works correctly
- test-add2.pixel – tests that float addition works correctly
- test-add3.pixel – tests that matrix addition works correctly
- test-arith1.pixel – tests that arithmetic operators and their precedence work
- test-float1pixel – tests that the printf function works correctly
- test-for1.pixel – tests that the for loop works correctly
- test-func1.pixel – tests that calling a function within another function works correctly
- test-if1.pixel – tests that an if statement that is true will work correctly
- test-if2.pixel – tests that an if and else statement will work correctly
- test-mult1.pixel – tests that integer multiplication works correctly
- test-while1.pixel – tests that the while loop works correctly

7 Lessons Learned

7.1 Alex Anthony Cortes-Ose

I now feel as though I fully understand the lifecycle of a programming language. From every step, struggling through this class, the material, the assignments, and especially the project, I felt as though I can build anything. Team collaboration is key and especially work distribution in every stage of the process. Make sure to test often and early, and that a language is fully functional before it is perfect. This allows proper progress to be made!

7.2 Dillon Davis

I learned so much about how a compiler functions and links together to perform the ‘magic’ that I once believed a compiler did. Understanding the flow of information from the scanner, parser, ast, semant, sast, codegen and built in functions really ties together a developers understanding of higher-level coding. I know that when I first began my computer science journey at Columbia, it was very strange to me that there were built in functions within a language that I could regularly use. I didn’t quite grasp how that was my first semester as a Freshman, and I simply accepted the ‘magic’ for built-in functions and many other things the compiler handles. I also greatly improved my functional programming skills through many hours of working on these files, and while I do not think I am a pro by any means I know I am much more comfortable with functional programming and its data flow. This also led to further improving my ability to implement recursive functions to solve problems working with higher-level programming, and I may start thinking about ways to link ocaml to my JavaScript projects which I know is professor Edward’s favorite language.

This project requires team-work, sacrifice and initiative. I learned that it is extremely important to be dedicated to the group of people you are on a team with, and it is important to be cognizant of not only one’s self but their teammates as well. If someone says that they will be joining an office hours meeting or request time out a group members day to get help understanding concepts, priorities become evident if one does not show up or communicate conflicts within a reasonable manner to value the other person’s time. It is extremely important to communicate struggles to teammates if someone needs help understanding material, but it is not another person’s responsibility to teach the material while balancing other classes and life situations as well. I learned that self-discipline, responsibility, communication and accountability are the foundations for a successful team, and I believe that we could have done better in these categories.

I greatly appreciate PLT for opening my eyes to a different side of programming that was completely unknown to me prior, and I will continue to build up my toolkit as I graduate and begin my career.

7.3 Jessica Kim

Throughout this project, I learned a lot about the general process it takes to create a programming language. I was glad I could familiarize myself with OCaml outside of assignments and lectures. This project was definitely one that was very hands-on and required a lot of thinking. Much teamwork was also required, and I think our team was able to work together very well. Apart from OCaml, I was also able to brush up on version control, which I

thought was useful. I hope I will be able to contribute to similar projects in the future and apply what I have learned from this class as well as from working on Pixel.

7.4 Jessica Peng

I think that accounting for unplanned events and starting early is a really important factor. Before the second part of the project was due I received a concussion which hurt the progress of the project but my teammates were able to carry on. A way we could have prevented the consequences of these unplanned events is starting earlier and planning the exact tasks needed to be completed rather than waiting for the day or two before to rely on consecutive all-nighters to grind.

I also think that events became extremely hectic during midterm and finals season and therefore it becomes harder and harder to meet in order to work on the project since everyone has back-to-back exams and assignment deadlines. I also learned that working “in-person”, or “on-call” together, is 100% more effective and efficient than working on certain parts outside individually. Not only is it a motivating factor to see your peers working alongside you, but a lot of times it became difficult to advance sections I was coding because I needed to understand the tools we were using or what functions we wanted exactly, but it was not as efficient just texting and waiting for a response.

Lastly, I think scheduling and organization is probably the most important factor that contributes to the success of this project. It is crucial to establish early the expected deadlines and deliverables in the very beginning of the project so we can split up tasks and work towards a goal, instead of dispensing energy not making progress. We had a few meetings where we were just trying to figure out exactly what was due and what we needed and those helped define a more concrete goal to work towards, but it may have been better executed if we did this since the beginning.

Overall, I think the concept of creating a language is very interesting, and I enjoyed learning the “behind-the-scenes” of doing so with OCAML and C++. However, I wish that I could have taken this course in a non-virtual semester and been able to meet up and connect more with my team and TA’s/professor and get more out of the class.

8 Appendix

8.1 Scanner

```
(* Ocamllex scanner for Pixel *)
{ open Parser }

let digit = ['0' - '9']
let digits = digit+

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| "//"     { singlecomment lexbuf}
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| '['      { LBRACK }
| ']'      { RBRACK }
| ','      { COMMA }
| '.'      { DOT }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "**"     { EXP }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "fun"    { FUN }
| "::"     { DBLCOLON }
| "matrix" { MATRIX }
| "image"  { IMAGE }
```

```
| "while"  { WHILE }
| "return" { RETURN }
| "int"    { INT }
| "float"   { FLOAT }
| "void"   { VOID }
| "grayscale" { GRAYSCALE }
| "red"    { RED }
| "green"  { GREEN }
| "blue"   { BLUE }
| "rows"   { ROWS }
| "cols"   { COLS }
| digits as lxm { LITERAL(int_of_string lxm) }
| digits '.'  digit* ( ['e' 'E'] ['+' '-']? digits )? as lxm { FLIT(lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*     as lxm { ID(lxm) }
| '"' ([^ '"']* as lxm) '"' { STR_LITERAL(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and singlecomment = parse
  "\n" { token lexbuf }
| _    { singlecomment lexbuf }

and comment = parse
  "*/" { token lexbuf }
| _    { comment lexbuf }
```

8.2 Parser

```
/* Ocamyacc parser for Pixel */
%{
  open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK COMMA DOT
%token PLUS MINUS TIMES DIVIDE ASSIGN EXP
%token EQ LT LEQ GT GEQ IF FOR WHILE FUN DBLCOLON
%token NOT NEQ AND OR ELSE RETURN
%token INT FLOAT MATRIX IMAGE VOID STRING
%token GRAYSCALE RED GREEN BLUE ROWS COLS
%token <int> LITERAL
```

```
%token <string> ID FLIT STR_LITERAL
%token EOF

%start program
%type <Ast.program> program

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left EXP
%left PLUS MINUS
%left TIMES DIVIDE
%right NOT

%%

program:
  decls EOF { List.rev $1 }

decls:
    /* nothing */ { [] }
  | decls fdecl   { FuncDecl($2)::$1 }

fdecl:
  FUN DBLCOLON typ ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
  { {
    typ = $3;
    fname = $4;
    formals = $6;
    body = List.rev $9
  } }

formals_opt:
    /* nothing */ { [] }
  | formal_list   { $1 }

formal_list:
```

```
    typ ID                   { [($1, $2)] }
  | formal_list COMMA typ ID { ($3, $4) :: $1 }

typ:
    INT    { Int }
  | FLOAT  { Float }
  | MATRIX { Matrix }
  | IMAGE  { Image }
  | VOID   { Void }
  | STRING { String }

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    expr SEMI           { Expr $1 }
  | RETURN expr_opt SEMI { Return $2 }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
                                        { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
  | typ ID SEMI { Variable($1, $2, Noexpr) }
  | typ ID ASSIGN expr SEMI { Variable($1, $2, Assign($2, $4)) }
  | MATRIX DBLCOLON typ ID ASSIGN expr SEMI
                                        { MatrixAssign($3, $4, $6) }
  | ID LBRACK expr COMMA expr RBRACK ASSIGN expr SEMI
                               { MatrixAccessAssign($1, $3, $5, $8) }

expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }

expr:
    LITERAL { Literal($1) }
  | FLIT { Fliteral($1) }
  | ID { Id($1) }
  | STR_LITERAL { StrLiteral($1) }
  | LBRACK matrix_body RBRACK { MLiteral($2) }
```

```
  | expr PLUS expr { Binop($1, Add, $3) }
  | expr MINUS expr { Binop($1, Sub, $3) }
  | expr TIMES expr { Binop($1, Mult, $3) }
  | expr DIVIDE expr { Binop($1, Div, $3) }
  | expr EXP expr { Binop($1, Exp, $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq, $3) }
  | expr LT     expr { Binop($1, Less, $3) }
  | expr LEQ    expr { Binop($1, Leq, $3) }
  | expr GT     expr { Binop($1, Greater, $3) }
  | expr GEQ    expr { Binop($1, Geq, $3) }
  | expr AND    expr { Binop($1, And, $3) }
  | expr OR     expr { Binop($1, Or, $3) }
  | MINUS expr %prec NOT { Unop(Neg, $2) }
  | NOT expr { Unop(Not, $2) }
  | ID LPAREN args_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }
  | ID ASSIGN expr   { Assign($1, $3) }
  | ID DOT RED  { ImageRedAccess($1) }
  | ID DOT GREEN  { ImageGreenAccess($1) }
  | ID DOT BLUE  { ImageBlueAccess($1) }
  | ID DOT ROWS { MatrixRows($1) }
  | ID DOT COLS { MatrixCols($1) }
  | ID DOT GRAYSCALE { ImageGrayscaleAccess($1) }
  | ID LBRACK expr COMMA expr RBRACK { MatrixAccess($1, $3, $5) }

args_opt:
    /* nothing */ { [] }
  | args_list     { List.rev $1 }

args_list:
    expr                { [$1] }
  | args_list COMMA expr { $3 :: $1 }

matrix_body:
    LBRACK args_list RBRACK { [MLiteral(List.rev $2)] }
  | LBRACK args_list RBRACK COMMA matrix_body { MLiteral(List.rev $2) :: $5 }
```

8.3 Abstract syntax tree

```
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq
        | And | Or | Exp

type uop = Neg | Not

type typ = Int | Float | Void | String | Image | Matrix

type expr =
    Literal of int
  | Fliteral of string
  | MLiteral of expr list * int * int
  | StrLiteral of string
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of string * expr
  | Call of string * expr list
  | Noexpr
  | MatrixAccess of string * expr * expr
  | ImageRedAccess of string
  | ImageGreenAccess of string
  | ImageBlueAccess of string
  | ImageGrayscaleAccess of string
  | MatrixRows of string
  | MatrixCols of string

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
  | Variable of typ * string * expr
  | MatrixAssign of typ * string * expr
  | MatrixAccessAssign of string * expr * expr * expr (*Incomplete*)

type bind = typ * string * expr

type func_decl = {
  typ: typ;
```

```
  fname: string;
  formals: bind list;
  body: stmt list;
}

type decls =
  | StmtDecl of stmt
  | FuncDecl of func_decl

type program = decls list (*Go over how this differs from  `bind list * func_decl list`*)

(* Pretty printing *)
let string_of_typ = function
    Int -> "int"
  | Float -> "float"
  | Void -> "void"
  | String -> "string"
  | Image -> "image"
  | Matrix -> "matrix"

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"
  | Exp -> "**"

let string_of_uop = function
    Neg -> "-"
  | Not -> "!"

let rec string_of_expr = function
    Literal(l) -> string_of_int l
```

```
  | Fliteral(l) -> l
  | StrLiteral(s) -> "\"" ^ s ^ "\""
  | Id(s) -> s
  | MLiteral(l, _, _) -> "[" ^ String.concat ", " (List.map string_of_expr l) ^ "]"
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""
  | MatrixAccess(v, e1, e2) -> v ^ "[" ^ string_of_expr e1 ^ ", " ^ string_of_expr e2 ^ "]"
  | ImageRedAccess(v) -> v ^ ".red"
  | ImageGreenAccess(v) -> v ^ ".green"
  | ImageBlueAccess(v) -> v ^ ".blue"
  | ImageGrayscaleAccess(v) -> v ^ ".grayscale"
  | MatrixRows(v) -> v ^ ".rows"
  | MatrixCols(v) -> v ^ ".cols"

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ "; " ^ string_of_expr e2 ^ "; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
  | Variable(t, v, e) -> if string_of_expr e = ""
      then string_of_typ t ^ " " ^ v ^ ";\n" (* No value assigned *)
      else string_of_typ t ^ " " ^ string_of_expr e ^ ";\n" (* w/ value assigned *)
  | MatrixAssign(t, v, e) -> "matrix::" ^ string_of_typ t ^ " " ^ v ^ " = " ^ string_of_expr e ^
";\n"
  | MatrixAccessAssign(v, e1, e2, e3) -> v ^ "[" ^ string_of_expr e1 ^ ", " ^ string_of_expr e2 ^
"] = " ^ string_of_expr e3 ^ ";"

let string_of_vdecl (t, id, e) = if string_of_expr e = ""
  then string_of_typ t ^ " " ^ id ^ ";\n"
```

```
  else string_of_typ t ^ " " ^ string_of_expr e ^ ";\n"

let string_of_fdecl fdecl =
  "fun::" ^ string_of_typ fdecl.typ ^ " " ^ fdecl.fname ^ "(" ^
    String.concat ", " (List.map string_of_vdecl fdecl.formals) ^
  ")\n{\n" ^ String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_decl = function
  | StmtDecl(s) -> string_of_stmt s
  | FuncDecl(f) -> string_of_fdecl f

let string_of_program decls =
  String.concat "" (List.map string_of_decl decls) ^ "\n"
```

8.4 Static-semantic checker

```
(* Semantic checking for the Pixel compiler *)

open Ast
open Sast

module StringMap = Map.Make(String)

(* Semantic checking of the AST. Returns an SAST if successful,
   throws an exception if something is wrong.

   Check each global variable, then check each function *)

let check functions =

  (* Verify a list of bindings has no void types or duplicate names *)
  let check_binds (kind : string) (binds : bind list) =
    List.iter (function
        (Void, b, _) -> raise (Failure ("illegal void " ^ kind ^ " " ^ b))
      | _ -> ()) binds;
    let rec dups = function
        [] -> ()
      | ((_,n1,_) :: (_,n2,_) :: _) when n1 = n2 ->
          raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
```

```
    | _ :: t -> dups t
  in dups (List.sort (fun (_,a,_) (_,b,_) -> compare a b) binds)
in

(**** Check global variables ****)

(* check_binds "global" globals; *)

(**** Check functions ****)

(* Collect function declarations for built-in functions: no bodies *)
let built_in_decls =
  let add_bind map (name, ty) = StringMap.add name {
    typ = Void;
    fname = name;
    formals = [(ty, "x")];
    body = [] } map
  in List.fold_left add_bind StringMap.empty [ ("print", Int);
                                               ("printf", Float);
                               ("image_in", [String; String], Image);
                               ("image_out", [String; Image; String], Void);
                               ("convolute", [Matrix; Matrix], Matrix);
                               ("join", [Matrix; Matrix; Matrix], Image);
                               ("join", [Matrix], Image) ]
in

(* Add function name to symbol table *)
let add_func map fd =
  let built_in_err = "function " ^ fd.fname ^ " may not be defined"
  and dup_err = "duplicate function " ^ fd.fname
  and make_err er = raise (Failure er)
  and n = fd.fname (* Name of the function *)
  in match fd with (* No duplicate functions or redefinitions of built-ins *)
       _ when StringMap.mem n built_in_decls -> make_err built_in_err
     (* | _ when StringMap.mem n map -> make_err dup_err   *)
     | _ ->  StringMap.add n fd map
in

(* Collect all function names into one symbol table *)
let function_decls = List.fold_left add_func built_in_decls functions
in
```

```
  (* Return a function from our symbol table *)
  let find_func s =
    try StringMap.find s function_decls
    with Not_found -> raise (Failure ("unrecognized function " ^ s))
  in

  let _ = find_func "main" in (* Ensure "main" is defined *)

  let check_function func =
    (* Make sure no formals are void or duplicates *)
    check_binds "formal" func.formals;

    (* Raise an exception if the given rvalue type cannot be assigned to
       the given lvalue type *)
    let check_assign lvaluet rvaluet err =
       if lvaluet = rvaluet then lvaluet else raise (Failure err)
    in

    let rec get_dims = function
        MatrixLit l -> List.length l :: get_dims (List.hd l)
      | _ -> []
    in
    (* Raise an exception if dimensions of Matrix are not balanced *)
    let rec flatten d = function
      [] -> []
      | MatrixLit hd::tl -> if List.length hd != List.hd d then raise (Failure("Invalid dims"))
else List.append (flatten (List.tl d) hd) (flatten d tl)
      | a -> a
    in

    (* Build local symbol table of variables for this function *)
    let symbols = List.fold_left (fun m (ty, name) -> StringMap.add name ty m)
                      StringMap.empty (globals @ func.formals )
    in

    (* Return a variable from our local symbol table *)
    let type_of_identifier s =
      try StringMap.find s symbols
      with Not_found -> raise (Failure ("undeclared identifier " ^ s))
    in
```

```
    let matrix_access_type = function
        (_, _) -> Float
      | _ -> raise (Failure ("Illegal matrix access"))
    in

    (* Return a semantically-checked expression, i.e., with a type *)
    let rec expr symbols = function
        Literal  l -> (Int, SLiteral l)
      | Fliteral l -> (Float, SFliteral l)
      | SLiteral l -> (String, SSLiteral)
      | Noexpr     -> (Void, SNoexpr)
      | Id s       -> (type_of_identifier s, SId s)
      | MLiteral l ->
          let d = get_dims (MLiteral l) in
          let rec all_match = function
            [] -> ignore()
            | hd::tl -> if tl != [] then
                          let (t1, _) = expr hd in let (t2, _) = expr (List.hd tl) in
                          if t1 = t2 then all_match tl else raise (Failure ("Data Mismatch in
MLiteral: " ^ string_of_typ t1 ^ " does not match " ^ string_of_typ t2))
                        else ignore()
          in
          all_match l;
          if List.length d > 2 then (Matrix, SMLiteral ((List.map expr l), List.hd d, List.hd
(List.tl d)))
          else if List.length d = 2 then (Matrix, SMLiteral ( (List.map expr (flatten (List.tl d)
l)), List.hd d, List.hd (List.tl d)))
          else if List.length d = 1 then (Matrix, SMLiteral ( (List.map expr (flatten (List.tl d)
l)), List.hd d, 1))
          else (Matrix, SMLiteral ( (List.map expr l), 0,0))
      | MatrixAccess(v, e1, e2) ->
          let _ = (match (expr symbols e1) with
                Int -> Int
              | _ -> raise (Failure ("Attempting to access with a non-integer type")))
            and _ = (match (expr symbols e2) with
                Int -> Int
              | _ -> raise (Failure ("Attempting to access with a non-integer type")))
            in matrix_access_type (type_of_identifier v symbols)
      | ImageRedAccess v ->
          if StringMap.mem v symbols
```

```
      then if (type_of_identifier v symbols) = Image then Matrix
      else raise (Failure ("Cannot call .red on non image datatype"))
    else raise (Failure ("Image variable does not exist"))
| ImageGreenAccess (v) ->
    if StringMap.mem v symbols
      then if (type_of_identifier v symbols) = Image then Matrix
      else raise (Failure ("Cannot call .green on non image datatype"))
    else raise (Failure ("Image variable does not exist"))
| ImageBlueAccess (v) ->
    if StringMap.mem v symbols
      then if (type_of_identifier v symbols) = Image then Matrix
      else raise (Failure ("Cannot call .blue on non image datatype"))
    else raise (Failure ("Image variable does not exist"))
| ImageGrayscaleAccess (v) ->
    if StringMap.mem v symbols
      then if (type_of_identifier v symbols) = Image then Matrix
      else raise (Failure ("Cannot call .grayscale on non image datatype"))
    else raise (Failure("Image variable does not exist"))
| MatrixRows (v) ->
    if StringMap.mem v symbols
      then if (type_of_identifier v symbols) = Matrix then Int
      else raise (Failure ("Cannot call .rowsize on non matrix datatype"))
    else raise (Failure ("Matrix variable does not exist"))
| MatrixCols (v) ->
    if StringMap.mem v symbols
      then if (type_of_identifier v symbols) = Matrix then Int
      else raise (Failure ("Cannot call .colsize on non matrix catatype"))
    else raise (Failure ("Matrix variable does not exist"))
| Assign(var, e) as ex ->
    let lt = type_of_identifier var
    and (rt, e') = expr e in
    let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
      string_of_typ rt ^ " in " ^ string_of_expr ex
    in (check_assign lt rt err, SAssign(var, (rt, e')))
| Unop(op, e) as ex ->
    let (t, e') = expr e in
    let ty = match op with
      Neg when t = Int || t = Float -> t
    | Not when t = Bool -> Bool
    | _ -> raise (Failure ("illegal unary operator " ^
                           string_of_uop op ^ string_of_typ t ^
```

```
                              " in " ^ string_of_expr ex))
          in (ty, SUnop(op, (t, e')))
      | Binop(e1, op, e2) as e ->
          let t1 = expr e1
          and t2 = expr e2 in
          (* All binary operators require operands of the same type *)
          let same = t1 = t2 in
          let t1_dim = (match t1 with
            (m,n) -> (m, n)
            | _ -> (-1, -1)
          ) in
          let t2_dim = (match t2 with
            (m,n) -> (m, n)
            | _ -> (-1, -1)
          )
          (* in let t1_dim = match t1 with *)

          (* Determine expression type based on operator and operand types *)
          in (match op with
            Add | Sub | Mult | Div when same && t1 = Int   -> Int
          | Add | Sub | Mult | Div when same && t1 = Float -> Float
          | Equal | Neq            when same               -> Int
          | Less | Leq | Greater | Geq
                     when same && (t1 = Int || t1 = Float) -> Int
          | And | Or when same && t1 = Int -> Int
          (* matrix op matrix *)
          (* | Add when t1 = (hd t1_dim, tl t1_dim)
                       && t2 = (hd t2_dim, tl t2_dim)
                       if same then (hd t1_dim, tl t1_dim)
                       else raise (Failure ("Cannot add/subtract matrices with different
dimensions")) *)
          | Add when same && t1 = (hd t1_dim, tl t1_dim) && t2 = (hd t2_dim, tl t2_dim) ->
                    (hd t1_dim, tl t1_dim)
                  (* else raise (Failure ("Cannot add/subtract matrices with different
dimensions")) *)
          | Mult when t1 = (hd t1_dim, tl t1_dim)
                  && t2 = (hd t2_dim, tl t2_dim) ->
                  if (tl t1_dim = hd t2_dim) then (hd t1_dim, tl t2_dim)
                  else raise (Failure ("Matrices cannot be multiplied given their dimensions"))
          (* matrix op scalar *)
          | Add | Exp | Mult when not same
```

```
                              && t1 = (hd t1_dim, tl t1_dim) && ((t2 = Int) || (t2 = Float))
                                -> (hd t1_dim, tl t1_dim)
                              (* else raise (Failure ("Scalar ops with matrices can only use ints
or floats")) *)
          (* Scalar op matrix *)
          | Add | Exp | Mult when not same
                              && t2 = hd t2_dim, tl t2_dim ->
                              if ((t1 = Int) || (t1 = Float)) then (hd t2_dim, tl t2_dim)
                              else raise (Failure ("Scalar ops with matrices can only use ints or
doubles"))
          | _ -> raise (Failure ("illegal binary operator")))
             (* Failure ("illegal binary operator " ^
                       string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
                       string_of_typ t2 ^ " in " ^ string_of_expr e))
          in (ty, SBinop((t1, e1'), op, (t2, e2'))) *)
      | Call(fname, args) as call ->
          let fd = find_func fname in
          let param_length = List.length fd.formals in
          if List.length args != param_length then
            raise (Failure ("expecting " ^ string_of_int param_length ^
                            " arguments in " ^ string_of_expr call))
          else let check_call (ft, _) e =
            let (et, e') = expr e in
            let err = "illegal argument found " ^ string_of_typ et ^
              " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr e
            in (check_assign ft et err, e')
          in
          let args' = List.map2 check_call fd.formals args
          in (fd.typ, SCall(fname, args'))
    in

    let check_bool_expr e =
      let (t', e') = expr e
      and err = "expected Boolean expression in " ^ string_of_expr e
      in if t' != Int then raise (Failure err) else (t', e')
    in

    (* Return a semantically-checked statement i.e. containing sexprs *)
    let rec check_stmt = function
        Expr e -> SExpr (expr e)
      | If(p, b1, b2) -> SIf(check_bool_expr p, check_stmt b1, check_stmt b2)
```

```
| For(e1, e2, e3, st) ->
   SFor(expr e1, check_bool_expr e2, expr e3, check_stmt st)
| While(p, s) -> SWhile(check_bool_expr p, check_stmt s)
| Return e -> let (t, e') = expr e in
  if t = func.typ then SReturn (t, e')
  else raise (
   Failure ("return gives " ^ string_of_typ t ^ " expected " ^
            string_of_typ func.typ ^ " in " ^ string_of_expr e))

     (* A block is correct if each statement is correct and nothing
        follows any Return statement.  Nested blocks are flattened. *)
| Block sl ->
    let rec check_stmt_list = function
        [Return _ as s] -> [check_stmt s]
      | Return _ :: _   -> raise (Failure "nothing may follow a return")
      | Block sl :: ss  -> check_stmt_list (sl @ ss) (* Flatten blocks *)
      | s :: ss         -> check_stmt s :: check_stmt_list ss
      | []              -> []
    in SBlock(check_stmt_list sl)

| Variable (typ, varname, e) as call ->
    if StringMap.mem varname symbols then
      (ignore (expr symbols e) ; symbols)
    else
      let expr_type = expr symbols e
      in if expr_type = Void then
        check_stmt (StringMap.add varname expr_type symbols) call
      else if typ = Matrix then
        check_stmt (StringMap.add varname expr_type symbols) call (* Adds the dimmat *)
      else if typ = expr_type then
        check_stmt (StringMap.add varname expr_type symbols) call
      else raise (Failure ("Local var type does not match"))
| MatrixAssign (typ, varname, e) as call ->
    if StringMap.mem varname symbols then
      (ignore (expr symbols e) ; symbols)
    else
      let expr_type = expr symbols e in
      if expr_type = Matrix && typ = Float then
        check_stmt (StringMap.add varname expr_type symbols) call
        (* SMatrixAssign(typ, varname, expr e, hd get_dims expr e, tl get_dims expr e) *)
      else raise (Failure ("Local var type does not match"))
```

```
            (* else if typ = Float then
              check_stmt (StringMap.add varname expr_type symbols) call (* Adds the dimmat *)
            else if typ = expr_type then
              check_stmt (StringMap.add varname expr_type symbols) call *)
      | MatrixAccessAssign (mat_name, idx_row, idx_col, vall) as call ->
          if StringMap.mem mat_name symbols then
          (ignore (expr symbols vall) ; symbols)
            else
              let expr1_type = expr symbols idx_row
              in let expr2_type = expr symbols idx_col
              in let expr3_type = expr symbols vall in
              if (expr1_type = Int && expr2_type = Int && expr3_type = Float)
              then check_stmt (StringMap.add mat_name expr3_type symbols) call
              else
              raise (Failure ("Invalid type for Matrix Access Assignment"))

    in (* body of check_function *)
    { styp = func.typ;
      sfname = func.fname;
      sformals = func.formals;
      sbody = match check_stmt (Block func.body) with
        SBlock(sl) -> sl
      | _ -> raise (Failure ("internal error: block didn't become a block?"))
    }
  in List.map check_function functions
```

8.5 Code generation

```
(* Code generation: translate takes a semantically checked AST and
produces LLVM IR

LLVM tutorial: Make sure to read the OCaml version of the tutorial

http://llvm.org/docs/tutorial/index.html

Detailed documentation on the OCaml LLVM library:

http://llvm.moe/
http://llvm.moe/ocaml/
```

```
*)

module L = Llvm
module A = Ast
open Sast

module StringMap = Map.Make(String)

(* translate : Sast.program -> Llvm.module *)
let translate decls =
  let context    = L.global_context () in
  let llmem = L.MemoryBuffer.of_file "util.o" in
  let llm = Llvm_bitreader.parse_bitcode context llmem in

  (* Create the LLVM compilation module into which
     we will generate code *)
  let the_module = L.create_module context "Pixel" in

  (* Get types from the context *)
  let i32_t      = L.i32_type    context
  and i8_t       = L.i8_type     context
  and float_t     = L.double_type context
  and void_t     = L.void_type   context
  and string_t   = L.pointer_type (L.i8_type context)
  and image_t   = L.pointer_type (
    match L.type_by_name llm "struct.image" with
        None -> raise (Failure "missing struct Image")
      | Some t -> t
  )
  and matrix_t   = L.pointer_type (
    match L.type_by_name llm "struct.matrix" with
        None -> raise (Failure "missing struct Matrix")
      | Some t -> t
  )
  in

  (* Return the LLVM type for a Pixel type *)
  let ltype_of_typ = function
      A.Int    -> i32_t
    | A.Float  -> float_t
    | A.Void   -> void_t
```

```
    | A.String -> string_t
    | A.Image  -> image_t
    | A.Matrix -> matrix_t
  in

  (* Define built-in function types *)
  let printf_t : L.lltype =
      L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
  let printf_func : L.llvalue =
      L.declare_function "printf" printf_t the_module in

  (* let initMatrix_t = L.function_type matrix_t [| float_t; i32_t; i32_t |] in *)
  let initMatrix_t = L.function_type matrix_t [| L.pointer_type (L.pointer_type float_t); i32_t;
i32_t; |] in
  let initMatrix_f = L.declare_function "initMatrix" initMatrix_t the_module in
  let access_t = L.function_type float_t [| matrix_t; i32_t; i32_t |] in
  let access_f = L.declare_function "access" access_t the_module in
  let accessAssign_t = L.function_type void_t [| matrix_t; i32_t; i32_t; float_t |] in
  let accessAssign_f = L.declare_function "accessAssign" accessAssign_t the_module in
  let image_in_t = L.function_type image_t [| string_t; string_t |] in
  let image_in_f = L.declare_function "image_in" image_in_t the_module in
  let image_out_t = L.function_type void_t [| string_t; image_t; string_t |] in
  let image_out_f = L.declare_function "image_out" image_out_t the_module in
  let convolute_t = L.function_type matrix_t [| matrix_t; matrix_t |] in
  let convolute_f = L.declare_function "convolute" convolute_t the_module in
  let join_color_t = L.function_type image_t [| matrix_t; matrix_t; matrix_t |] in
  let join_color_f = L.declare_function "join" join_color_t the_module in
  let join_grayscale_t = L.function_type image_t [| matrix_t; |] in
  let join_grayscale_f = L.declare_function "join" join_grayscale_t the_module in
  let multiply_matrix_t = L.function_type matrix_t [| matrix_t; matrix_t |] in
  let multiply_matrix_f = L.declare_function "multiply_matrix" multiply_matrix_t the_module in
  let add_matrix_t = L.function_type matrix_t [| matrix_t; matrix_t |] in
  let add_matrix_f = L.declare_function "add_matrix" add_matrix_t the_module in
  let scale_matrix_t = L.function_type matrix_t [| matrix_t; float_t |] in
  let scale_matrix_f = L.declare_function "scale_matrix" scale_matrix_t the_module in
  let exp_matrix_t = L.function_type matrix_t [| matrix_t; float_t |] in
  let exp_matrix_f = L.declare_function "exp_matrix" exp_matrix_t the_module in

  (* Define each function (arguments and return type) so we can
     call it even before we've created its body *)
  let function_decls : (L.llvalue * sfunc_decl) StringMap.t =
```

```
  let function_decl m fdecl =
    let name = fdecl.sfname
    and formal_types =
           Array.of_list (List.map (fun (t,_, _) -> ltype_of_typ t) fdecl.sformals)
    in let ftype = L.function_type (ltype_of_typ fdecl.styp) formal_types in
    StringMap.add name (L.define_function name ftype the_module, fdecl) m in
  List.fold_left function_decl StringMap.empty decls
in

(* Fill in the body of the given function *)
let build_function_body fdecl =
  let (the_function, _) = StringMap.find fdecl.sfname function_decls in
  let builder = L.builder_at_end context (L.entry_block the_function) in

  let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder in
  let float_format_str = L.build_global_stringptr "%g\n" "fmt" builder in
  let string_format_str = L.build_global_stringptr "%s\n" "fmt" builder in
  (* Construct the function's "locals": formal arguments and locally
     declared variables.  Allocate each on the stack, initialize their
     value, if appropriate, and remember their values in the "locals" map *)
  (* let local_vars =
    let add_formal m (t, n) p =
      L.set_value_name n p;
         let local = L.build_alloca (ltype_of_typ t) n builder in
      ignore (L.build_store p local builder);
         StringMap.add n local m

    (* Allocate space for any locally declared variables and add the
     * resulting registers to our map *)
    and add_local m (t, n) =
      let local_var = L.build_alloca (ltype_of_typ t) n builder
      in StringMap.add n local_var m
    in

    let formals = List.fold_left2 add_formal StringMap.empty fdecl.sformals
        (Array.to_list (L.params the_function)) in
    List.fold_left add_local formals fdecl.slocals
  in *)
  let local_vars =
    let add_formal m (t, n, _) p =
      L.set_value_name n p;
```

```
    let local = L.build_alloca (ltype_of_typ t) n builder in
      ignore (L.build_store p local builder);
  StringMap.add n local m

    and add_local m (t, n) =
    (* add_local m (t, n) = *)
      let local_var = L.build_alloca (ltype_of_typ t) n builder
      in StringMap.add n local_var m
    in

  let formals = List.fold_left2 add_formal StringMap.empty fdecl.sformals
    (Array.to_list (L.params the_function))
  (* in *)
  in List.fold_left add_local formals []
in

  (* Return the value for a variable or formal argument *)
  let lookup n =
    try StringMap.find n local_vars
    with Not_found -> raise (Failure "Undefined variable ")
  in

  (* Construct code for an expression; return its value *)
  (*
    | SBinop of sexpr * op * sexpr
    | SUnop of uop * sexpr
    | SMatrixAccess of string * sexpr * expr
    | SImageRedAccess of string
    | SImageGreenAccess of string
    | SImageBlueAccess of string
    | SImageGrayscaleAccess of string
    | SMatrixRows of string
    | SMatrixCols of string
  *)
  (* let rec expr builder ((_, e) : sexpr) local_vars = match e with *)
  let rec expr builder ((_, e) : sexpr) lvars = match e with
           SLiteral i  -> L.const_int i32_t i
    | SFliteral l -> L.const_float_of_string float_t l
    | SStrLiteral l -> L.build_global_stringptr l "tmp" builder
    | SNoexpr     -> L.const_int i32_t 0
    | SId s       -> L.build_load (lookup s) s builder
```

```
      | SAssign (s, e) -> let value = StringMap.find s local_vars
          in let e' = expr builder e local_vars
          in ignore(L.build_store e' value builder);
          e'
      | SMLiteral (contents, rows, cols) ->
          let rec expr_list = function
            [] -> []
            | hd::tl -> (expr builder hd local_vars)::expr_list tl
          in
          let contents' = expr_list contents
          in
          (* let m = L.build_call matrix_init_f [| L.const_int i32_t cols; L.const_int i32_t rows
|] "matrix_init" builder *)
          let m = L.build_call initMatrix_f [| L.pointer_type (L.pointer_type float_t) contents';
L.const_int i32_t rows; L.const_int i32_t cols |] "initMatrix" builder
          in
          ignore(List.map (fun v -> L.build_call store_matrix_f [| m ; v |] "store_val" builder)
contents'); m
      | SBinop ((A.Float, _) as e1, op, e2) ->
        let e1' = expr builder e1 local_vars
        and e2' = expr builder e2 local_vars in
        (match op with
          A.Add     -> L.build_fadd
        | A.Sub     -> L.build_fsub
        | A.Mult    -> L.build_fmul
        | A.Div     -> L.build_fdiv
        | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
        | A.Neq     -> L.build_fcmp L.Fcmp.One
        | A.Less    -> L.build_fcmp L.Fcmp.Olt
        | A.Leq     -> L.build_fcmp L.Fcmp.Ole
        | A.Greater -> L.build_fcmp L.Fcmp.Ogt
        | A.Geq     -> L.build_fcmp L.Fcmp.Oge
        (* | A.Exp     -> L. *)
        | A.And | A.Or ->
            raise (Failure "internal error: semant should have rejected and/or on float")
        ) e1' e2' "tmp" builder
      (* | SBinop ((A.Matrix, A.Matrix) as e1, op, e2) ->
        let e1' = expr builder e1 local_vars
        and e2' = expr builder e2 local_vars in
        (match op with
```

```
          A.Add     -> L.build_call add_matrix_f [| (expr builder e1 local_vars); (expr builder
e2 local_vars) |] "add_matrix" builder
        | A.Mult    -> L.build_call multiply_matrix_f [| (expr builder e1 local_vars); (expr
builder e2 local_vars) |] "multiply_matrix" builder
        ) e1' e2' "tmp" builder
      (* | SBinop ((A.Matrix, A.Float) as e1, op, e2) | SBinop ((A.Matrix, A.Float) as e1, op,
e2) -> *)
        | SBinop ((A.Matrix, A.Float) as e1, op, e2) ->
        let e1' = expr builder e1 local_vars
        and e2' = expr builder e2 local_vars in
        (match op with
          A.Mult  -> L.build_call scale_matrix_f [| (expr builder e1 local_vars); (expr builder
e2 local_vars) |] "scale_matrix" builder
        | A.Exp   -> L.build_call exp_matrix_f [| (expr builder e1 local_vars); (expr builder e2
local_vars) |] "exp_matrix" builder
        ) e1' e2' "tmp" builder *)
      | SBinop (e1, op, e2) ->
        let e1' = expr builder e1 local_vars
        and e2' = expr builder e2 local_vars in
        (match op with
          A.Add     -> L.build_add
        | A.Sub     -> L.build_sub
        | A.Mult    -> L.build_mul
        | A.Div     -> L.build_sdiv
        | A.And     -> L.build_and
        | A.Or      -> L.build_or
        | A.Equal   -> L.build_icmp L.Icmp.Eq
        | A.Neq     -> L.build_icmp L.Icmp.Ne
        | A.Less    -> L.build_icmp L.Icmp.Slt
        | A.Leq     -> L.build_icmp L.Icmp.Sle
        | A.Greater -> L.build_icmp L.Icmp.Sgt
        | A.Geq     -> L.build_icmp L.Icmp.Sge
        ) e1' e2' "tmp" builder
      | SUnop(op, ((t, _) as e)) ->
        let e' = expr builder e local_vars in
        (match op with
          A.Neg when t = A.Float -> L.build_fneg
        | A.Neg                  -> L.build_neg
        | A.Not                  -> L.build_not
        ) e' "tmp" builder
```

```
      | SMatrixAccess (s, e1, e2) -> L.build_call access_f [| (expr builder s local_vars); (expr
builder e1 local_vars); (expr builder e2 local_vars) |] "access" builder
      | SImageRedAccess (s) -> let img_val = StringMap.find s local_vars
                               in let pointer_to_red = L.build_struct_gep img_val 0 "red" builder
                               in (L.build_load pointer_to_red "red" builder)
      | SImageGreenAccess (s) -> let img_val = StringMap.find s local_vars
                               in let pointer_to_green = L.build_struct_gep img_val 1 "green"
builder
                               in (L.build_load pointer_to_green "green" builder)
      | SImageBlueAccess (s) -> let img_val = StringMap.find s local_vars
                               in let pointer_to_blue = L.build_struct_gep img_val 2 "blue"
builder
                               in (L.build_load pointer_to_blue "blue" builder)
      | SImageGrayscaleAccess (s) -> let img_val = StringMap.find s local_vars
                               in let pointer_to_grayscale = L.build_struct_gep img_val 3
"grayscale" builder
                               in (L.build_load pointer_to_grayscale "grayscale" builder)
      | SMatrixRows (s) -> let mat_val = StringMap.find s local_vars
                               in let pointer_to_rows = L.build_struct_gep mat_val 0 "rows"
builder
                               in (L.build_load pointer_to_rows "rows" builder)
      | SMatrixCols (s) -> let mat_val = StringMap.find s local_vars
                               in let pointer_to_cols = L.build_struct_gep mat_val 1 "cols"
builder
                               in (L.build_load pointer_to_cols "cols" builder)
      (* Match built-in function names *)
      | SCall ("print", [e]) ->
            L.build_call printf_func [| int_format_str ; (expr builder e local_vars) |]
            "printf" builder
      | SCall ("printf", [e]) ->
            L.build_call printf_func [| float_format_str ; (expr builder e local_vars) |]
            "printf" builder
      | SCall ("image_in", [e1; e2]) ->
        L.build_call image_in_f [| (expr builder e1 local_vars) ; (expr builder e2 local_vars) |]
        "image_in" builder
      | SCall ("image_out", [e1; e2; e3]) ->
        L.build_call image_out_f [| (expr builder e1 local_vars) ; (expr builder e2 local_vars) ;
(expr builder e3 local_vars) |]
        "image_out" builder
      | SCall ("convolute", [e1; e2]) ->
```

```
        L.build_call image_out_f [| (expr builder e1 local_vars) ; (expr builder e2 local_vars)
|]
        "convolute" builder
      | SCall ("join", [e1; e2; e3]) ->
        L.build_call join_grayscale_f [| (expr builder e1 local_vars) |]
        "join" builder
      | SCall ("join", [e1]) ->
        L.build_call join_color_f [| (expr builder e1 local_vars) ; (expr builder e2 local_vars)
; (expr builder e3 local_vars) |]
        "join" builder
      | SCall (f, args) ->
        let (fdef, fdecl) = StringMap.find f function_decls in
          (* let llargs = List.rev (List.map (expr builder) (List.rev args)) in *)
          let llargs = List.rev (List.map ( fun x -> expr builder x local_vars ) (List.rev args))
in
          let result = (match fdecl.styp with
                         A.Void -> ""
                       | _ -> f ^ "_result") in
          L.build_call fdef (Array.of_list llargs) result builder
        in

      (* LLVM insists each basic block end with exactly one "terminator"
        instruction that transfers control.  This function runs "instr builder"
        if the current block does not already have a terminator.  Used,
        e.g., to handle the "fall off the end of the function" case. *)
      let add_terminal builder instr =
        match L.block_terminator (L.insertion_block builder) with
          Some _ -> ()
        | None -> ignore (instr builder)
      in

    (* Build the code for the given statement; return the builder for
       the statement's successor (i.e., the next instruction will be built
       after the one generated by this call) *)
    let rec stmt builder lvals = function
            SBlock sl -> List.fold_left stmt builder local_vars sl
      | SExpr e -> ignore(expr builder e local_vars); builder
      | SVariable (t, n, e) ->
        (* local_vars := List.fold_left add_local local_vars [(t, n)]
        if e != Noexpr then (
          let e' = expr builder e local_vars in ignore(L.build_store e' (lookup s) builder); e'
```

```
        ) *)
        let local_var = L.build_alloca (ltype_of_typ typ) name builder
        in let new_local_vars = StringMap.add name local_var local_vars
        in ignore (L.build_store (fst (expr builder e new_local_vars)) local_var builder);
        (builder, new_local_vars)
      | SMatrixAssign (t, s, e, rows, cols) ->
          let local_var = L.build_alloca (ltype_of_typ t) s builder
          in let new_local_vars = StringMap.add s local_var local_vars
          in ignore (L.build_store (fst (expr builder (L.build_call initMatrix_f [| (expr builder
e local_vars); (expr builder e1 local_vars); (expr builder e2 local_vars) |] "initMatrix"
builder) new_local_vars)) local_var builder);
          (builder, new_local_vars)
      | SMatrixAccessAssign (s, e1, e2, e3) ->
        (* LOOK HERE *)
        L.build_call accessAssign_f [| (expr builder s local_vars); (expr builder e1 local_vars);
(expr builder e2 local_vars); (expr builder e3 local_vars) |] "accessAssign" builder
      | SReturn e -> ignore(match fdecl.styp with
                            (* Special "return nothing" instr *)
                            A.Void -> L.build_ret_void builder
                            (* Build return statement *)
                          | _ -> L.build_ret (expr builder e local_vars) builder );
                    builder
      | SIf (predicate, then_stmt, else_stmt) ->
        let bool_val = expr builder predicate local_vars in
        let merge_bb = L.append_block context "merge" the_function in
        let build_br_merge = L.build_br merge_bb in (* partial function *)

          let then_bb = L.append_block context "then" the_function in
          add_terminal (stmt (L.builder_at_end context then_bb) local_vars then_stmt)
            build_br_merge;

          let else_bb = L.append_block context "else" the_function in
          add_terminal (stmt (L.builder_at_end context else_bb) local_vars else_stmt)
            build_br_merge;

        ignore(L.build_cond_br bool_val then_bb else_bb builder);
        L.builder_at_end context merge_bb

      | SWhile (predicate, body) ->
        let pred_bb = L.append_block context "while" the_function in
        ignore(L.build_br pred_bb builder);
```

```
        let body_bb = L.append_block context "while_body" the_function in
        add_terminal (stmt (L.builder_at_end context body_bb) local_vars body)
          (L.build_br pred_bb);

        let pred_builder = L.builder_at_end context pred_bb in
          let bool_val = expr pred_builder predicate local_vars in

            let merge_bb = L.append_block context "merge" the_function in
            ignore(L.build_cond_br bool_val body_bb merge_bb pred_builder);
          L.builder_at_end context merge_bb

          (* Implement for loops as while loops *)
      | SFor (e1, e2, e3, body) -> stmt builder local_vars
          ( SBlock [SExpr e1 ; SWhile (e2, SBlock [body ; SExpr e3]) ] )
        in

        (* Build the code for each statement in the function *)
        let builder = stmt builder local_vars (SBlock fdecl.sbody) in

        (* Add a return if the last block falls off the end *)
        add_terminal builder (match fdecl.styp with
            A.Void -> L.build_ret_void
          | A.Float -> L.build_ret (L.const_float float_t 0.0)
          | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
  in

  List.iter build_function_body decls;
  the_module
```

8.6 Standard library (util.cpp)

```
#define OPENCV_TRAITS_ENABLE_DEPRECATED

#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/imgcodecs.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <iostream>
#include <string>
#include <math.h>
```

```
#include <stdlib.h>
#include <stdio.h>

using namespace cv;

static void die(const char *message)
{
    perror(message);
    exit(1);
}

struct matrix {
    int rows;
    int cols;
    float** matrixAddr;
};
struct image {
    matrix* red;
    matrix* green;
    matrix* blue;
    matrix* grayscale;
};
float** getMat (Mat mat) {
    float** matrixValues = (float**) malloc(mat.rows * sizeof(float*));
    for (int i = 0; i < mat.rows; i++) {
        float* matrix_row = (float*) malloc(mat.cols * sizeof(float));
        *(matrixValues + i) = matrix_row;
        for (int j = 0; j < mat.cols; j++) {
            matrix_row[j] = mat.at<float>(i, j);
        }
    }
    return matrixValues;
}
float* get1D (float** matrix2d, int n, int m) {
    float* b;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            b[i * m + j] = matrix2d[i][j];
        }
    }
    return b;
```

```
}
matrix* initMatrix(float* data, int num_rows, int num_cols) {
    Mat newMatrix = Mat::zeros(num_rows, num_cols, CV_32F);
    for (int i = 0; i < (num_rows * num_cols); i++) {
        newMatrix.at<float>(i) = data[i];
    }

    matrix* result = (matrix*) malloc(sizeof(struct matrix));
    result->cols = num_cols;
    result->rows = num_rows;
    result->matrixAddr = getMat(newMatrix);

    return result;
}
float access(matrix* src, int row, int col) {
    return src->matrixAddr[row][col];
}
void accessAssign(matrix* src, int row, int col, float value) {
    src->matrixAddr[row][col] = value;
}
// image_in         : String filePath, String mode            -> image
image* image_in (String filePath, String mode) {
    Mat rgb, g;
    if (mode == "COLOR") {
        imread(filePath, IMREAD_COLOR).convertTo(rgb, CV_32F, 1, 0);
        imread(filePath, IMREAD_GRAYSCALE).convertTo(g, CV_32F, 1, 0);

        if (!rgb.data || !g.data) {
            die("Error processing image");
        }
        std::cout << "Finished loading\n" << std::endl;

        std::vector<Mat> p;
        split(rgb, p);

        Mat nr = Mat::zeros(p[2].rows, p[2].cols, CV_32F);
        for (int i = 0; i < p[2].rows; i++) {
            for (int j = 0; j < p[2].cols; j++) {
                nr.at<float>(i, j) = p[2].at<float>(i, j);
            }
        }
```

```
        Mat ng = Mat::zeros(p[1].rows, p[1].cols, CV_32F);
        for (int i = 0; i < p[1].rows; i++) {
            for (int j = 0; j < p[1].cols; j++) {
                ng.at<float>(i, j) = p[1].at<float>(i, j);
            }
        }
        Mat nb = Mat::zeros(p[0].rows, p[0].cols, CV_32F);
        for (int i = 0; i < p[0].rows; i++) {
            for (int j = 0; j < p[0].cols; j++) {
                nb.at<float>(i, j) = p[0].at<float>(i, j);
            }
        }
        Mat ngs = Mat::zeros(g.rows, g.cols, CV_32F);
        for (int i = 0; i < g.rows; i++) {
            for (int j = 0; j < g.cols; j++) {
                ngs.at<float>(i, j) = g.at<float>(i, j);
            }
        }

        image* result = (image*) malloc(sizeof(struct image));
        std::cout << "Loaded memory for image struct\n" << std::endl;
        result->red = initMatrix(get1D(getMat(nr), nr.rows, nr.cols), nr.rows, nr.cols);
        std::cout << "Saved red\n" << std::endl;
        result->green = initMatrix(get1D(getMat(ng), ng.rows, ng.cols), ng.rows, ng.cols);
        std::cout << "Saved green\n" << std::endl;
        result->blue = initMatrix(get1D(getMat(nb), nb.rows, nb.cols), nb.rows, nb.cols);
        std::cout << "Saved blue\n" << std::endl;
        result->grayscale = initMatrix(get1D(getMat(ngs), ngs.rows, ngs.cols), ngs.rows,
ngs.cols);
        std::cout << "Saved grayscale\n" << std::endl;
        return result;
    } else if (mode == "GRAYSCALE") {
        imread(filePath, IMREAD_GRAYSCALE).convertTo(g, CV_32F, 1, 0);

        if (!g.data) {
            die("Error processing image");
        }
        std::cout << "Finished loading\n" << std::endl;
        image* result = (image*) malloc(sizeof(struct image));
        std::cout << "Loaded memory for image struct\n" << std::endl;
        result->grayscale = initMatrix(get1D(getMat(g), g.rows, g.cols), g.rows, g.cols);
```

```
        std::cout << "Saved grayscale\n" << std::endl;

        return result;
    } else {
        die("invalid color mode: <COLOR, GRAYSCALE>");
    }
}
// image_out        : String filePath, image img, String mode -> void
void image_out (String filePath, image* img, String mode) {
    if (mode == "COLOR") {
        Mat dst  = Mat::zeros(img->red->rows, img->red->cols, CV_32F);
        Mat final = Mat::zeros(img->red->rows, img->red->cols, CV_32F);

        std::vector<Mat> channels;
        Mat nr = Mat::zeros(img->red->rows, img->red->cols, CV_32F);
        for (int i = 0; i < img->red->rows; i++) {
            for (int j = 0; j < img->red->cols; j++) {
                nr.at<float>(i, j) = img->red->matrixAddr[i][j];
            }
        }
        Mat ng = Mat::zeros(img->green->rows, img->green->cols, CV_32F);
        for (int i = 0; i < img->green->rows; i++) {
            for (int j = 0; j < img->green->cols; j++) {
                ng.at<float>(i, j) = img->green->matrixAddr[i][j];
            }
        }
        Mat nb = Mat::zeros(img->blue->rows, img->blue->cols, CV_32F);
        for (int i = 0; i < img->blue->rows; i++) {
            for (int j = 0; j < img->blue->cols; j++) {
                nb.at<float>(i, j) = img->blue->matrixAddr[i][j];
            }
        }

        channels.push_back(nb);
        channels.push_back(ng);
        channels.push_back(nr);

        std::cout << "start merge" << std::endl;
        merge(channels, dst);

        imwrite(filePath, dst);
```

```
    } else if (mode == "GRAYSCALE") {
        Mat dst  = Mat::zeros(img->grayscale->rows, img->grayscale->cols, CV_32F);
        Mat final = Mat::zeros(img->grayscale->rows, img->grayscale->cols, CV_32F);
        Mat GS = Mat::zeros(img->grayscale->rows, img->grayscale->cols, CV_32F);
        for (int i = 0; i < img->grayscale->rows; i++) {
            for (int j = 0; j < img->grayscale->cols; j++) {
                GS.at<float>(i, j) = img->grayscale->matrixAddr[i][j];
            }
        }
        std::cout << GS << std::endl;
        imwrite(filePath, GS);
    } else {}
}
// join (color)     : matrix red, matrix blue, matrix green  -> image
image* join (matrix* red, matrix* green, matrix* blue) {
    Mat R = Mat::zeros(red->rows, red->cols, CV_32F);
    for (int i = 0; i < red->rows; i++) {
        for (int j = 0; j < red->cols; j++) {
            R.at<float>(i, j) = red->matrixAddr[i][j];
        }
    }
    Mat G = Mat::zeros(green->rows, green->cols, CV_32F);
    for (int i = 0; i < green->rows; i++) {
        for (int j = 0; j < green->cols; j++) {
            G.at<float>(i, j) = green->matrixAddr[i][j];
        }
    }
    Mat B = Mat::zeros(blue->rows, blue->cols, CV_32F);
    for (int i = 0; i < blue->rows; i++) {
        for (int j = 0; j < blue->cols; j++) {
            B.at<float>(i, j) = blue->matrixAddr[i][j];
        }
    }

    image* result = (image*) malloc(sizeof(struct image));
    result->red = initMatrix(get1D(getMat(R), R.rows, R.cols), R.rows, R.cols);
    result->green = initMatrix(get1D(getMat(G), G.rows, G.cols), G.rows, G.cols);
    result->blue = initMatrix(get1D(getMat(B), B.rows, B.cols), B.rows, B.cols);
    std::cout << "finished creating joined color image" << std::endl;
    return result;
}
```

```
// join (grayscale) : matrix grayscale                        -> image
image* join (matrix* grayscale) {
    Mat GS = Mat::zeros(grayscale->rows, grayscale->cols, CV_32F);
    for (int i = 0; i < grayscale->rows; i++) {
        for (int j = 0; j < grayscale->cols; j++) {
            GS.at<float>(i, j) = grayscale->matrixAddr[i][j];
        }
    }

    image* result = (image*) malloc(sizeof(struct image));
    result->grayscale = initMatrix(get1D(getMat(GS), GS.rows, GS.cols), GS.rows, GS.cols);
    return result;
}
// multiply_matrix  : matrix A, matrix B                      -> matrix
matrix* multiply_matrix (matrix* lhs, matrix* rhs) {
    Mat A = Mat::zeros(lhs->rows, lhs->cols, CV_32F);
    for (int i = 0; i < lhs->rows; i++) {
        for (int j = 0; j < lhs->cols; j++) {
            A.at<float>(i, j) = lhs->matrixAddr[i][j];
        }
    }
    Mat B = Mat::zeros(rhs->rows, rhs->cols, CV_32F);
    for (int i = 0; i < rhs->rows; i++) {
        for (int j = 0; j < rhs->cols; j++) {
            B.at<float>(i, j) = rhs->matrixAddr[i][j];
        }
    }

    Mat C = A * B;

    matrix* result = (matrix*) malloc(sizeof(struct matrix));
    result->cols = rhs->cols;
    result->rows = lhs->rows;
    result->matrixAddr = getMat(C);
    return result;
}
// add_matrix       : matrix A, matrix B                      -> matrix
matrix* add_matrix (matrix* lhs, matrix* rhs) {
    Mat A = Mat::zeros(lhs->rows, lhs->cols, CV_32F);
    for (int i = 0; i < lhs->rows; i++) {
        for (int j = 0; j < lhs->cols; j++) {
```

```
            A.at<float>(i, j) = lhs->matrixAddr[i][j];
        }
    }
    Mat B = Mat::zeros(rhs->rows, rhs->cols, CV_32F);
    for (int i = 0; i < rhs->rows; i++) {
        for (int j = 0; j < rhs->cols; j++) {
            B.at<float>(i, j) = rhs->matrixAddr[i][j];
        }
    }

    Mat C = A + B;

    matrix* result = (matrix*) malloc(sizeof(struct matrix));
    result->cols = rhs->cols;
    result->rows = lhs->rows;
    result->matrixAddr = getMat(C);
    return result;
}
// scale_matrix     : matrix A, float s                      -> matrix
matrix* scale_matrix (matrix* m, float s) {
    Mat A = Mat::zeros(m->rows, m->cols, CV_32F);
    for (int i = 0; i < m->rows; i++) {
        for (int j = 0; j < m->cols; j++) {
            A.at<float>(i, j) = m->matrixAddr[i][j];
        }
    }

    Mat C = A * s;

    matrix* result = (matrix*) malloc(sizeof(struct matrix));
    result->cols = m->cols;
    result->rows = m->rows;
    result->matrixAddr = getMat(C);
    return result;
}
// exp_matrix       : matrix A, float exp                    -> matrix
matrix* exp_matrix (matrix* m, float exp) {
    Mat A = Mat::zeros(m->rows, m->cols, CV_32F);
    for (int i = 0; i < m->rows; i++) {
        for (int j = 0; j < m->cols; j++) {
            A.at<float>(i, j) = pow(m->matrixAddr[i][j], exp);
```

```
        }
    }

    matrix* result = (matrix*) malloc(sizeof(struct matrix));
    result->cols = m->cols;
    result->rows = m->rows;
    result->matrixAddr = getMat(A);
    return result;
}
// convolute        : matrix src, matrix kernel              -> matrix
// Mat_<float> convolute(const Mat_<float>& src, const Mat_<float>& kernel) {
matrix* convolute(matrix* s, matrix* k) {
    Mat src = Mat::zeros(s->rows, s->cols, CV_32F);
    for (int i = 0; i < s->rows; i++) {
        for (int j = 0; j < s->cols; j++) {
            src.at<float>(i, j) = s->matrixAddr[i][j];
        }
    }
    Mat kernel = Mat::zeros(k->rows, k->cols, CV_32F);
    for (int i = 0; i < k->rows; i++) {
        for (int j = 0; j < k->cols; j++) {
            kernel.at<float>(i, j) = k->matrixAddr[i][j];
        }
    }
    Mat dst(src.rows, src.cols, CV_32F);

    Mat flipped_kernel(kernel.rows, kernel.cols, CV_32F);
    flip(kernel, flipped_kernel, -1);

    const int dx = kernel.cols / 2;
    const int dy = kernel.rows / 2;

    for (int i = 0; i<src.rows; i++)
    {
        for (int j = 0; j<src.cols; j++)
        {
            float tmp = 0.0f;
            for (int k = 0; k<flipped_kernel.rows; k++)
            {
              for (int l = 0; l<flipped_kernel.cols; l++)
              {
```

```
                int x = j - dx + l;
                int y = i - dy + k;
                if (x >= 0 && x < src.cols && y >= 0 && y < src.rows)
                    tmp += src.at<float>(y, x) * flipped_kernel.at<float>(k, l);
              }
            }
            dst.at<float>(i, j) = saturate_cast<float>(tmp);
        }
    }
    Mat final = Mat::zeros(dst.rows, dst.cols, CV_32F);
    for (int i = 0; i < dst.rows; i++) {
        for (int j = 0; j < dst.cols; j++) {
            final.at<float>(i, j) = dst.at<float>(i, j);
        }
    }
    matrix* result = (matrix*) malloc(sizeof(struct matrix));
    result->cols = final.cols;
    result->rows = final.rows;
    result->matrixAddr = getMat(final);
    return result;
}
matrix* zeros(int rows, int cols) {
    Mat zm = Mat::zeros(rows, cols, CV_32F);
    matrix* result = (matrix*) malloc(sizeof(struct matrix));
    result->cols = cols;
    result->rows = rows;
    result->matrixAddr = getMat(zm);
    return result;
}

int main() {
    //SOBEL HORIZONTAL
    //---
    float floatsGS[] = { 1, 1, 1, 0, 0, 0, -1, -1, -1 };
    matrix* kernel = initMatrix(floatsGS, 3, 3);
    image* a = image_in("./steam-engine.png", "GRAYSCALE");
    matrix* cvgs = convolute(a->grayscale, kernel);
    image* final = join(cvgs);
    image_out("./FROMCPP.png", final, "GRAYSCALE");

    //BLUR COLOR
```

```
//----
// float floatsC[] = { 1, 1, 1, 1, 1, 1, 1, 1, 1 };
// matrix* kernel = initMatrix(floatsC, 3, 3);
// image* a = image_in("./cat.png", "COLOR");
// matrix* cvr = scale_matrix(convolute(a->red, kernel), 0.11);
// matrix* cvg = scale_matrix(convolute(a->green, kernel), 0.11);
// matrix* cvb = scale_matrix(convolute(a->blue, kernel), 0.11);
// image* final = join(cvr, cvg, cvb);
// image_out("./FROMCPP.png", final, "COLOR");

//MATRIX SCALAR
//---
// float floatsM[] = {1, 1, 1, 1, 1, 1, 1, 1, 1};
// matrix* base = initMatrix(floatsM, 3, 3);
// matrix* final = scale_matrix(base, 3);
// for (int i = 0; i < final->rows; i++) {
//     for (int j = 0; j < final->cols; j++) {
//         std::cout << final->matrixAddr[i][j] << " ";
//     }
// }
// std::cout << std::endl;

// MATRIX ADDITION
// ---
// float floatsM[] = {1, 1, 1, 1, 1, 1, 1, 1, 1};
// matrix* A = initMatrix(floatsM, 3, 3);
// matrix* B = initMatrix(floatsM, 3, 3);
// matrix* final = add_matrix(A, B);
// for (int i = 0; i < final->rows; i++) {
//     for (int j = 0; j < final->cols; j++) {
//         std::cout << final->matrixAddr[i][j] << " ";
//     }
// }
// std::cout << std::endl;

// MATRIX MULTIPLICATION
// ---
// float floatsA[] = {-1, 4, 2, 3};
// float floatsB[] = {9, -3, 6, 1};
// matrix* A = initMatrix(floatsA, 2, 2);
// matrix* B = initMatrix(floatsB, 2, 2);
```

```
// matrix* final = multiply_matrix(A, B);
// for (int i = 0; i < final->rows; i++) {
//     for (int j = 0; j < final->cols; j++) {
//         std::cout << final->matrixAddr[i][j] << " ";
//     }
// }
// std::cout << std::endl;

// MATRIX EXPONENTS
// ---
// float floatsM[] = {2, 2, 2, 2};
// matrix* base = initMatrix(floatsM, 2, 2);
// matrix* final = exp_matrix(base, 3);
// for (int i = 0; i < final->rows; i++) {
//     for (int j = 0; j < final->cols; j++) {
//         std::cout << final->matrixAddr[i][j] << " ";
//     }
// }
// std::cout << std::endl;

// MATRIX ACCESS
// ---
// float matA[] = {1, 1, 1, 1, 1, 1, 1, 1, 1};
// matrix* A = initMatrix(matA, 3, 3);
// std::cout << access(A, 0, 0) << std::endl;

// MATRIX ACCESS ASSIGN
// ---
// float matA[] = {1, 1, 1, 1, 1, 1, 1, 1, 1};
// matrix* A = initMatrix(matA, 3, 3);
// accessAssign(A, 0, 0, 200);
// std::cout << access(A, 0, 0) << std::endl;

// ZERO MATRIX
// ---
// matrix* A = zeros(3, 3);
// for (int i = 0; i < A->rows; i++) {
//     for (int j = 0; j < A->cols; j++) {
//         std::cout << A->matrixAddr[i][j] << " ";
//     }
// }
```

```
    // std::cout << std::endl;
}
```

8.7 Tests

```
==> fail-assign1.pixel <==
fun::void main() {

  int i;
  float f;
  str s;

  i = 10;
  f = 5.0;
  s = "Hello World";

  i = 3.0; /* Error: assigning a float to an integer */
}

==> fail-assign2.pixel <==
fun::void main() {

  int i;
  float f;
  str s;

  i = 10;
  f = 5.0;
  s = "Hello World";

  f = "3.0"; /* Error: assigning a string to a float */
}

==> fail-assign3.pixel <==
fun::void main() {

  int i;
  float f;
  str s;
```

```
  i = 10;
  f = 5.0;
  s = "Hello World";

  s = 3; /* Error: assigning an integer to a string */
}

==> fail-assign4.pixel <==
fun::void main() {

  dict::str,str d;
  matrix::int m;
  list::int l;

  d = {"name": "John Doe", "date":
  04-25-21"};
  m = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8]
  ];
  l = [9, 10, 11];

  d = [
    [12, 13],
    [14, 15]
  ]; /* Error: assigning a matrix to a dictionary */
}

==> fail-assign5.pixel <==
fun::void main() {

  dict::str,str d;
  matrix::int m;
  list::int l;

  d = {"name": "John Doe", "date":
  04-25-21"};
  m = [
    [0, 1, 2],
    [3, 4, 5],
```

```
    [6, 7, 8]
  ];
  l = [9, 10, 11];

  m = [12, 13]; /* Error: assigning a list to a matrix */
}

==> fail-assign6.pixel <==
fun::void main() {

  dict::str,str d;
  matrix::int m;
  list::int l;

  d = {"name": "John Doe", "date":
  04-25-21"};
  m = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8]
  ];
  l = [9, 10, 11];

  l = {"title": "Harry Potter", "author": "J. K. Rowling"}; /* Error: assigning a dictionary to a
list */
}

==> fail-dead1.pixel <==
float f = 5.0; /* Error: code outside a function */

fun::void main() {
  int i;
  i = 10;
}

==> fail-dead2.pixel <==
fun::void main() {
  int i;
  i = 10;
}
```

```
float f = 5.0; /* Error: code outside a function */

==> fail-dead3.pixel <==
fun::void main1() {
  int i;
  i = 10;
}

float f = 5.0; /* Error: code outside a function */

fun::void main2() {
  int j;
  j = 20;
}

==> fail-expr1.pixel <==
fun::void main() {

  int i;
  float f;
  str s;

  i = 10;
  f = 5.0;
  s = "Hello World";

  i + f; /* Error: int + float */
}

==> fail-expr2.pixel <==
fun::void main() {

  int i;
  float f;
  str s;

  i = 10;
  f = 5.0;
  s = "Hello World";

  i + s; /* Error: int + string */
```

```
}

==> fail-for1.pixel <==
fun::int main() {

  int i;

  for (i = 0; j < 10 ; i = i + 1) {} /* Error: j is undefined */

  return 0;

}

==> fail-for2.pixel <==
fun::int main() {

  int i;

  for (i = 0; i; i = i + 1) {} /* Error: i is an integer, not a boolean */

  return 0;

}

==> fail-for3.pixel <==
fun::int main() {

  int i;

  for (i = 0; i < 10 ; i = j + 1) {} /* Error: j is undefined */

  return 0;

}

==> fail-for4.pixel <==
fun::int main() {

  int i;

  for (i = 0; i < 10 ; i = i + 1) {
```

```
    foo(); /* Error: function foo does not exist */
  }

  return 0;

}

==> fail-for5.pixel <==
fun::int main() {

  int i;

  for (i = 0; i < 10 ; i++) {} /* Error: ++ operator not supported */

  return 0;

}

==> fail-func1.pixel <==
fun::int foo() {}

fun::int bar() {}

fun::int baz() {}

fun::void bar() {} /* Error: duplicate function bar */

fun::int main()
{
  return 0;
}

==> fail-func2.pixel <==
fun::int foo(int a, float b, int c) {}

fun::void bar(int a, float b, int a) {} /* Error: duplicate parameter a in bar */

fun::int main()
{
  return 0;
}
```

```
==> fail-func3.pixel <==
fun::int foo(int a, float b, int c) {}

fun::void bar(int a, void b, int c) {} /* Error: illegal void parameter b */

fun::int main()
{
  return 0;
}

==> fail-func4.pixel <==
fun::int foo() {}

fun::void bar() {}

fun::int join() {} /* Error: should not be able to define a built-in function */

fun::void baz() {}

fun::int main()
{
  return 0;
}

==> fail-func5.pixel <==
fun::int foo() {}

fun::int bar() {
  int a;
  void b; /* Error: illegal void local b */
  float c;

  return 0;
}

fun::int main()
{
  return 0;
}
```

```
==> fail-func6.pixel <==
fun::void foo(int a, float b)
{
}

fun::int main()
{
  foo(42, 22.2);
  foo(42); /* Error: wrong number of arguments */
}

==> fail-func7.pixel <==
fun::void foo(int a, float b)
{
}

fun::int main()
{
  foo(42, 22.2);
  foo(42, 22.2, 10); /* Error: wrong number of arguments */
}

==> fail-func8.pixel <==
fun::void foo(int a, float b)
{
}

fun::int main()
{
  foo(42, 22.2);
  foo(42, "hello"); /* Error: should be float argument, not string */
}

==> fail-if1.pixel <==
fun::int main()
{
  if (1) {}
  if (0) {} else {}
  if (42) {} /* Error: predicate should evaluate to 1 or 0 */
}
```

```
==> fail-if2.pixel <==
fun::int main()
{
  if (1) {
    foo; /* Error: undeclared variable */
  }
}

==> fail-if3.pixel <==
fun::int main()
{
  if (1) {
    return 42;
  } else {
    return bar; /* Error: undeclared variable */
  }
}

==> fail-matrix1.pixel <==
fun::void main() {

  matrix::str m;

  m = [
    ["hello", "world"],
    ["John", "Doe"]
  ]; /* Error: matrices cannot store strings */
}

==> fail-matrix2.pixel <==
fun::void main() {

  matrix::float m;

  m = [
    [5.0, 6.5, 3.3],
    [4.4, 11.]
  ]; /* Error: incorrect dimensions of a matrix */
}

==> fail-matrix3.pixel <==
```

```
fun::void main() {

  matrix::float m1 = [
    [1.0, 2.5, 3.],
    [4.2, 5.1, 6.7]
  ];

  matrix::float m2 = [
    [1., 1.],
    [2., 3.3]
  ];

  matrix::float m3;
  m3 = m1 + m2; /* Error: inconsistent dimension of matrices */

}

==> fail-nomain.pixel <==

==> fail-return1.pixel <==
fun::int main()
{
  return "val"; /* Error: should return int */
}

==> fail-return2.pixel <==
fun::void foo()
{
  if (1) {
    return 42; /* Error: should return void */
  } else {
    return;
  }
}

fun::int main()
{
  return 42;
}

==> fail-while1.pixel <==
```

```
fun::int main()
{
  int i;

  while (1) {
    i = i + 1;
  }

  while (42) { /* Error: predicate should evaluate to 1 or 0 */
    i = i + 1;
  }

}

==> fail-while2.pixel <==
fun::int main()
{
  int i;

  while (1) {
    i = i + 1;
  }

  while (1) {
    foo(); /* Error: foo is undefined */
  }

}

==> test-add1.pixel <==
fun::int add(int x, int y)
{
  return x + y;
}

fun::int main()
{
  print(add(17, 25)); /* Should return 42 */
  return 0;
}
```

```
==> test-add2.pixel <==
fun::float add(float x, float y)
{
  return x + y;
}

fun::float main()
{
  print(add(17.0, 25.5)); /* Should return 42.5 */
  return 0.0;
}

==> test-add3.pixel <==
fun::void main() {

  matrix::float m1 = [
    [1.0, 2.0, 3.0],
    [4.0, 5.0, 6.0]
  ];

  matrix::float m2 = [
    [1.0, 1.0, 1.0],
    [2.5, 3.5, 4.5]
  ];

  print(m1 + m2); /* Should print [[2.0, 3.0, 4.0], [6.5, 8.5, 10.5]] */

}

==> test-arith1.pixel <==
fun::int main()
{
  print(1 + 2 * 3 + 4); /* Should print 11 */
  return 0;
}

==> test-arith2.pixel <==
fun::int foo(int a)
{
  return a;
}
```

```
fun::int main()
{
  int a;
  a = 42;
  a = a + 5;
  print(a);
  return 0;
}

==> test-fib.pixel <==
fun::int fib(int x)
{
  if (x < 2) {
  return 1;
  }
  return fib(x-1) + fib(x-2);
}

fun::int main()
{
  print(fib(0));
  print(fib(1));
  print(fib(2));
  print(fib(3));
  print(fib(4));
  print(fib(5));
  return 0;
}

/* Should print:
1
1
2
3
5
8
*/

==> test-float1.pixel <==
fun::int main()
```

```
{
  float a;
  a = 3.14159267;
  printf(a); /* Should print 3.14159 */
  return 0;
}

==> test-float2.pixel <==
fun::int main()
{
  float a;
  float b;
  float c;
  a = 3.14159267;
  b = -2.71828;
  c = a + b;
  printf(c); /* Should print 0.423313 */
  return 0;
}

==> test-float3.pixel <==
fun::void testfloat(float a, float b)
{
  printf(a + b);
  printf(a - b);
  printf(a * b);
  printf(a / b);
  print(a == b);
  print(a == a);
  print(a != b);
  print(a != a);
  print(a > b);
  print(a >= b);
  print(a < b);
  print(a <= b);
}

fun::int main()
{
  float c;
  float d;
```

```
  c = 42.0;
  d = 3.14159;

  testfloat(c, d);

  testfloat(d, d);

  return 0;
}

/* Should print:
45.1416
38.8584
131.947
13.369
0
1
1
0
1
1
0
0
6.28318
0
9.86959
1
1
1
0
0
0
1
0
1
*/

==> test-for1.pixel <==
fun::int main()
{
```

```
  int i;
  for (i = 0 ; i < 5 ; i = i + 1) {
    print(i);
  }
  print(42);
  return 0;
}

/* Should print:
0
1
2
3
4
42
*/

==> test-for2.pixel <==
fun::int main()
{
  int i;
  i = 0;
  for ( ; i < 5; ) {
    print(i);
    i = i + 1;
  }
  print(42);
  return 0;
}

/* Should print:
0
1
2
3
4
42
*/

==> test-func1.pixel <==
fun::int add(int a, int b)
```

```
{
  int c;
  c = a + b;
  return c;
}

fun::int main()
{
  int d;
  d = add(52, 10);
  print(d); /* Should print 62 */
  return 0;
}

==> test-func2.pixel <==
fun::int foo(int a)
{
  return a;
}

fun::int main()
{
  return 0; /* Should not return anything */
}

==> test-func3.pixel <==
fun::void foo(int a)
{
  print(a + 3);
}

fun::int main()
{
  foo(40); /* Should print 43 */
  return 0;
}

==> test-if1.pixel <==
fun::int main()
{
  if (1) {
```

```
    print(42);
  }
  print(17);
  return 0;
}

/* Should print:
42
17
*/

==> test-if2.pixel <==
fun::int main()
{
  if (1) {
    print(42);
  } else {
    print(8);
  }
  print(17);
  return 0;
}

/* Should print:
42
17
*/

==> test-if3.pixel <==
fun::int main()
{
  if (0) {
    print(42);
  } else {
    print(8);
  }
  print(17);
  return 0;
}

/* Should print:
```

```
8
17
*/

==> test-mult1.pixel <==
fun::int multiply(int x, int y)
{
  return x * y;
}

fun::int main()
{
  print(multiply(2, 3)); /* Should return 6 */
  return 0;
}

==> test-mult2.pixel <==
fun::float multiply(float x, float y)
{
  return x * y;
}

fun::float main()
{
  print(multiply(3.0, 1.5)); /* Should return 4.5 */
  return 0.0;
}

==> test-mult3.pixel <==
fun::void main() {

  matrix::float m1 = [
    [1.0, 2.0, 3.0],
    [4.0, 5.0, 6.0]
  ];

  matrix::float m2 = [
    [1.0, 1.0],
    [2.5, 3.5],
    [2.0, 4.0]
  ];
```

```
  print(m1 * m2); /* Should print [[12.0, 20.0], [28.5, 45.5]] */

}

==> test-mult4.pixel <==
fun::void main() {

  matrix::float m = [
    [1.0, 2.0, 3.0],
    [4.0, 5.0, 6.0]
  ];

  float f = 2.0;

  print(m * f); /* Should print [[2.0, 4.0, 6.0], [8.0, 10.0, 12.0]] */

}

==> test-while1.pixel <==
fun::int main()
{
  int i;
  i = 5;
  while (i > 0) {
    print(i);
    i = i - 1;
  }
  print(42);
  return 0;
}

/* Should print:
5
4
3
2
1
42
*/
```

```
==> test-while2.pixel <==
fun::int foo(int a)
{
  int j;
  j = 0;
  while (a > 0) {
    j = j + 2;
    a = a - 1;
  }
  return j;
}

fun::int main()
{
  print(foo(7)); /* Should print 14 */
  return 0;
}
```