4115 Final Report

# MATRIXMANIA

Cindy Espinosa
Desu Imudia
Diego Prado
Sophie Reese-Wirpsa
Emily Ringel

# Contents

# 1 Introduction

## 1.1 Motivation

Our team wanted to implement a language that targets matrix based problems with familiar syntax and no additional stress on the programmer with regards to memory allocation or access. There were a few pillars of our language design that we want to highlight: flexibility with approach to problem solving, simplicity of syntax, and similarity to preceding languages.

We purposefully did not implement too many matrix functions to give our users the most freedom to be creative with how they apply the language to many different situations where a matrix manipulation may be useful. MatrixMania was inspired by previous languages such as Python, Java, and C that we have been taught previously during our time at Columbia. We tried to make it more approachable than C for a newer programmer but less forgiving than Python. We aimed for a Java-level language–one does not have to completely understand memory allocation and pointers to use our language but still must be attentive to details in the syntax. Our language will be simple for programmers to learn quickly if they already know other languages such as Java or Python. Our control flow and logic are very similar so new programmers to MatrixMania will have no trouble diving right in!

## 1.2 Language Description

MatrixMania is an imperative programming language that is designed to support matrix-based manipulation. The goal of our language is to ease the manipulation of matrices. MatrixMania was created for those wanting to work primarily with matrices. The language includes a matrix data type. MatrixMania allows for efficient linear algebra calculations and easy access to rows and columns in matrices. Our language is used to write programs to manipulate and solve matrices. Our language can be used to build algorithms to invert an n-by-n matrix or to multiply a matrix by a scalar number. These types of algorithms would benefit from many of the features in our language, such as the built in ability to multiply a matrix or part of a matrix by a primitive type value or syntax to make indexing more readable.

# 2 Language Tutorial

## 2.1 Setup

MatrixMania requires the installation of the OCaml llvm library and C. Then download the following:

```
sudo apt-get install -y ocaml m4 llvm opam
opam init
opam install llvm.3.6 ocamlfind
eval ‘opam config env
```

The compiler is called upon by using the matrixmania.native command and streaming in a file of the .mm format. Our Makefile also includes a command to run a specific test, as shown below.

```
make filename=test_determinant.mm test
```

## 2.2 Basic Syntax

MatrixMania is very similar to Java with regards to syntax.

Every MatrixMania program has a main method which is defined as follows:

```
def int main(){
}
```

Statements and expressions end with a semi-colon. Unlike in Java, variables must be declared and initialized at the same time. MatrixMania has 2 main variable types: ints and floats.

```
 int x = 0;
float y = 12.45;
```

MatrixMania allows users to declare and work with matrices of ints or matrices of floats. The rows of a matrix literal are separated by a semi-colon. Values in a row are separated by a comma.

```
 matrix<int> x = [1, 5, 7; 3, 4, 12];
matrix<float> y = [2.9, 6.3; 4.67, 9.1];
```

Comments (both multi-line and single line) are begun with /* and ended with */.

## 2.3 Sample Programs

### 2.3.1 GCD

Listing 1: tests/test_gcd.mm

```
/*gcd program to showcase our language's ability to compute common cs problems */
def int gcd(int x, int y){
   if (x == 0){
      return y;
   }
   while(x != y){
      if(x > y) {x = x - y;}
      else {y = y - x;}
   }
   return x;
}
def int main( ){
   print(gcd(3, 15));
   print(gcd(18,24));
   print(gcd(45,120));
   return 0;
}
```

### 2.3.2 Matrix Transpose

Listing 2: tests/test_transpose.mm

```
def void transpose(matrix<int> original, matrix<int> result){
    /* using built-in functions getRows and getColumns */
    int rows = getRows(original);
    int cols = getColumns(original);

    for(int i = 0; i < cols; i=i+1){
      for(int j = 0; j < rows; j=j+1){
            result[i, j] = original[j, i];
      }
    }
}

def int main(){
    /* declare matrix */
    matrix<int> A = [1, 2, 3;
                     4, 5, 6];
    printm(A);

    print(1111111111);

    /* declare result matrix */
    matrix<int> A_T = [0, 0;
                       0, 0;
                       0, 0];
    transpose(A, A_T);

    printm(A_T);

    return 0;
}
```

# 3 Language Reference Manual

## 3.1 Lexical Conventions

This section specifies the lexical elements and content of the MatrixMania language source code after processing.

### 3.1.1 Tokens

Included in MatrixMania are six types of tokens: identifiers, keywords, constants, expression operators, and other separators.

### 3.1.2 Comments

Comments begin with /* and end with */ such that any text between them will be ignored. Comments can be split across multiple lines. Comments do not nest.

```
/* single line comment */
/*
multi-line
comment
*/
```

### 3.1.3 Separators

A separator separates tokens. White space is a separator, but not a token. We have the following separator tokens:

| Separator | Name | Description |
|---|---|---|
| ( | Left parenthesis | Used for opening function arguments, statements, and loops. |
| ) | Right parenthesis | Used for closing function arguments, statements, and loops. |
| { | Left curly bracket | Part of opening block separator for functions |
| } | Right curly bracket | Part of closing block separator for functions |
| [ | Left square bracket | Part of opening matrix initialization and access |
| ] | Right square bracket | Part of closing matrix initialization and access |
| , | Comma | Used in matrices to separate items in a row |
| . | Period | Used in `floats` |
| ; | Semicolon | Used to end statements |

### 3.1.4 Identifiers

An identifier is a sequence of letters and digits for naming variables and functions. An identifier must begin with an alphabetic character. Upper and lower case letters are distinct since identifiers are case sensitive.

### 3.1.5 Keywords

The following identifiers are reserved for use as keywords:

| Keyword | Description |
|---|---|
| `main` | main function; code with main function will be executed when the executable file runs after compilation |
| `return` | return function value |
| `int` | int variable type |
| `float` | float variable type |
| `matrix` | matrix variable type |
| `if` | if part of if-else or if-elif-else statement(s) |
| `elif` | elif part of if-elif-else statement(s) |
| `else` | else part of if-else or if-elif-else statement(s) |
| `for` | for loop |
| `while` | while loop |
| `def` | function indicator |
| `void` | void function type for functions that do not provide a return result to caller |

### 3.1.6 Literals

**Integer** An integer literal, with data type `int`, is a sequence of digits.

**Floating** A floating literal, with data type `float`, consists of an integer part, a decimal point, a fraction part, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing.

## 3.2 Data Types

| Keyword | Description | Example |
|---|---|---|
| int | a signed integer with 32-bit size | int x = 2; |
| float | a floating point value with 64-bit size | float y = 5.5; |
| matrix | a two-dimensional matrix of integers or floating point values, presented in row order – i.e. sequential values on the same row are stored adjacent to each other. Size information, such as the number of rows and columns, is stored with the object | matrix<int> z = [1,2;3,4];<br>matrix<float> z = [1.5,2.4;3.2,4.7]; |
| void | an empty set of values; used as the type returned by functions that generate no value | def void function_name(){} |

## 3.3 Declarations

MatrixMania uses explicit typing in which all variables must be explicitly declared with their types.

**Type Specifiers** The type-names are int, float, matrix¡int¿, matrix¡float¿, and void. One type-name and one identifier-name are given in each declaration. Example:

| |
|---|
| int a = 5; |
| float b = 5.0; |
| matrix<int> c = [1,2;3,4]; |
| matrix<float> d = [1.0,2.0;3.0,4.0]; |
| def void func() |

## 3.4 Objects and lvalues

**Object** An object is a mutible region of storage.

**lvalue** An lvalue refers to an object that can be assigned to; it is an object that persists beyond a single expression. The left operand must be an lvalue expression.

## 3.5 Conversions

**Int → Float** All `ints` may be converted without loss of significance to `float` through implicit type casting. This can be done when assigning variables or using operations.

## 3.6 Operators and Precedence

An operator is a character(s) that represents an action. The specific details of the operand expressions are in the Expression section of the LRM. An operand is the data value that is to be manipulated or operated on.

**Operators**

| **Operator** | **Operator Type** | **Description** |
|---|---|---|
| = | Assignment | Value to the right of operator is stored in variable to left of operator. The left and right hand sides of the assignment operator must be of the same data type or the left hand side can be a float with the right hand side as an int. |
| * | Multiplication | Multiplies two values together. Multiplying two values of the same type returns a value of that type. Multiplying an int and a float returns a float. Multiplying a matrix of some type and an int or float returns a matrix of the same type. Multiplication with two matrices runs a matrix multiplication algorithm. See more in section 3.12.1. |
| / | Division | Divides the first value by the second value. Division with two floats or a float and an int produces a float. Division with two ints produces an int. Division cannot be used on matrices. |
| % | Modulus | Returns the remainder when the first value is divided by the second value. Both operands must have type int. |
| + | Addition | Adds two values together. Adding two values of the same type returns a value of that type. Adding an int and a float returns a float. A matrix cannot be added to a scalar. Matrices of different types cannot be added. See more in section 3.12.1. |
| - | Subtraction | Subtracts the second value from the first value. Subtracting with two values of the same type returns a value of that type. Subtracting with an int and a float returns a float. A matrix cannot be subtracted from a scalar, and vice versa. Matrices of different types cannot be subtracted. See more in section 3.12.1. |

| | | |
|---|---|---|
| < | Less than comparison | Returns a 1 if the first value is less than the second, else 0. Operands can be either ints or floats. |
| > | Greater than comparison | Returns a 1 if the first value is greater than the second, else 0. Operands can be either ints or floats. |
| <= | Less than or equal to comparison | Returns a 1 if the first value is less than or equal to the second, else 0. Operands can be either ints or floats. |
| >= | Greater than or equal to comparison | Returns a 1 if the first value is greater than or equal to the second, else 0. Operands can be either ints or floats. |
| == | Equal to comparison | Returns a 1 if the values are equivalent, else 0. Operands can either be matching types of any type or be an int and a float. |
| != | Not equal to comparison | Returns a 1 if the values are not equivalent, else 0. Operands can either be matching types of any type or be an int and a float. |
| && | Logical AND operator | Returns a 1 if both operands are 1, else 0. Operands must both be ints. |
| \|\| | Logical OR operator | Returns a 1 if at least one operand is 1, else 0. Operands must both be ints. |
| ! | Logical NOT operator | Returns a 1 if the input is 0, else 0. Operand must be an int. |
| - | Negation operator | Negates the given value. Operand must be an int or a float. |

**Operator Precedence** If there is more than one operator present in a single expression, operations are performed according to operator precedence. Operators that share the same precedence are evaluated according to associativity. Left-associative operators evaluate from left to right, while right-associative operators evaluate from right to left. All operators are left-associative, except the assignment operator (=) and not operator (!).

| **Precedence** | **Operators** |
|---|---|
| highest | `!` |
| | `*, /, %` |
| | `+, -` |
| | `>, <, >=, <=` |
| | `==, !=` |
| | `&&, \|\|` |
| lowest | `=` |

## 3.7 Expressions

### 3.7.1 Primary Expression

Function calls group left to right

**Identifier** Identifiers are primary expressions. Identifiers are symbols which name the language entities (listed in keywords of the lexical conventions section 2c). Its type is specified by its declaration.

**Literal** An integer or floating constant is a primary expression.

{**Expression**} A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

### 3.7.2 Unary Operators

**`!` expression** The result of the logical NOT operator `!` is 1 if the value of the expression is 0, and 0 if the value of the expression is non-zero. The type of the result is `int`. This operator is applicable only to `int`s.

**`-` expression** The result of the logical negation operator is the return of a negative of the given value. For example if the value is 7, applying the - to it would give -7. Operand must be an `int` or a `float`.

### 3.7.3 Multiplicative Operators

The multiplicative operators `*`, `/`, and `%` group left-to-right.

**expression `*` expression** The binary `*` operator indicates multiplication. If both operands are `int`, the result is `int`. If one is `int` and one `float`, the result is `float`. If both are `float`, the result is `float`. If both are `matrix` of the same primitive type, the result is a `matrix` of that type. If one is a `matrix` and one an `int` (or `float`), the result is a `matrix`.

**expression `/` expression** The binary `/` operator indicates division. The same type considerations as for `*` (multiplication) apply, besides matrices. A `matrix` cannot be divided.

**expression `%` expression** The binary `%` operator yields the remainder from the division of the first expression by the second. Both operands must be `int`, and the result is `int`. In the current implementation, the remainder has the same sign as the dividend.

### 3.7.4 Additive Operators

The additive operators `+` and `-` group left-to-right.

**expression `+` expression** The result is the sum of the expressions. If both operands are `int`, the result is `int`. If both are `float`, the result is `float`. If one is `int` and one is `float`, the result is `float`. If both are matrix of the same primitive type, the result is a matrix of that type.

**expression `-` expression** The result is the difference of the operands. If both operands are `int`, `float`, or `matrix` the same type considerations as for + (addition) apply.

### 3.7.5 Relational Operators

The operators $<$ (less than), $>$ (greater than), $<=$ (less than or equal to) and $>=$ (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. Operand conversion is exactly the same as for the `+` operator.

The relational operators group left-to-right.

**expression $<$ expression**

**expression $>$ expression**

**expression $<=$ expression**

**expression $>=$ expression**

### 3.7.6 Equality Operator

The `==` (equal to) and the `!=` (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus `"a < b == c < d"` is 1 whenever `a<b` and `c<d` have the same truth-value)

**expression `==` expression**

**expression `!=` expression**

### 3.7.7 Logical Operators

**expression `&&` expression** The `&&` operator returns 1 if both its operands are non-zero, 0 otherwise. Guarantees left-to-right evaluation. The operands need not have the same type, but each must have one of the fundamental types (`int` or `float`).

**expression || expression** The || operator returns 1 if either of its operands is non-zero, and 0 otherwise. Guarantees left-to-right evaluation. The operands need not have the same type, but each must have one of the fundamental types (`int` or `float`).

### 3.7.8 Assignment Operators

Group right-to-left. All require an lvalue as their left operand. The value is the value stored in the left operand after the assignment has taken place.

**lvalue = expression** The value of the expression replaces that of the object referred to by the lvalue. The operands need not have the same type, but both must be `int` or `float`. If operands are different types (`float` and `int`) the assignment takes place as expected, preceded by conversion of the expression on the right. When both operands are `int` or `float`, no conversion takes place; the value of the expression is simply stored into the object referred to by the lvalue.

## 3.8 Constant Expressions

A constant expression is an expression that contains only constants. A constant expression can be evaluated during compilation rather than at run time, and can be used in any place that a constant can occur. Some places may require expressions that evaluate to a constant (after case, array bounds, initializers) and may be connected by binary operators when involving integer constants.

## 3.9 Statements

### 3.9.1 Expression Statements

Most statements are expression statements and take the form:

| |
|---|
| *expression;* |

They can be assignments or function calls.

### 3.9.2 Conditional Statements

Options for the conditional statement are:

| |
|---|
| `if`(*expression*) { *statements* } |

| |
|---|
| `if`(*expression*) {*statements*} `else` {*statements*} |

| |
|---|
| `if`(*expression*) { *statements* }<br>`elif`(*expression*) {*statements*}<br>...<br>`else` {*statements*} |

The expression is evaluated and if it is non-zero, then the first substatement is executed. For the second case, if the first expression is zero then the second substatement is checked, if it is non-zero, then the substatement is executed. If the second statement is zero, then the substatement connected to the else block is executed. One can have as many `elif` blocks between the `if` block and the `else` block.

### 3.9.3 Iteration Statements

**While Statement** The while statement takes the form:

```
while(expression) { statements }
```

The statement is executed while the expression evaluates to a non-zero value. The test of the expression occurs before the execution of the statement.

**For Statement** The for statement takes the form:

```
for(expression-1; expression-2 expression-3) {statements}
```

Expression-1 is executed once before the execution of the statement in the code block. This expression specifies the initialization for the loop. Expression-2 defines the condition for executing the statement in the code block. The test of the condition occurs before each iteration and the loop is excited when the expression evaluates to 0. Expression-3 is executed every time after the statement in the code block has been executed. It typically specifies an incrementation of the value initialized in expression 1.

### 3.9.4 Execution Statements

**Return Statement** A function uses the return statement to return to its caller. The type of a return statement must match the type in the function declaration. It can take one of the following forms:

```
return;
return (expression);
```

## 3.10 Functions

MatrixMania allows for the usage of functions in our language. Functions can be used to separate subroutines of code. You must write the function definition to create and use a function. All executable programs in our language must have at least one function - the `main` function. Our language allows for recursive functions.

### 3.10.1 Function Declaration and Definition

Functions consist of a function header and a function body. The header contains the keyword `def` indicating that it is a function, a return type of the function, the name of the function, and an optional parameter list enclosed in parentheses. The function body is enclosed by a pair of curly braces.

```
def int add(int a, int b){
    int c = a + b;
    return c;
}
```

A function can return the following types: `int`, `float`, and `void`. Matrix types cannot be returned because they are passed by reference.

### 3.10.2 Function Calls

In order to be able to call a function, the function must have been declared already. Functions are called by using its name and supplying the necessary parameters. The function call will execute using the given parameters and return the value as defined by the function.

```
add(2, 3);
```

### 3.10.3 Main Function

Every program requires the main function in order to execute. The main function must be defined. The return type for main is always int. The ‘main’ function can be written to accept no parameters or to accept parameters from the command line.

```
def int main(){
    int first = 2;
    int second = 3;
    int third = add(first, second);
    print(third);
}
```

## 3.11 Scope Rules

Variables can be defined inside and outside of functions. For variables that are defined inside of a function, they are bound within the brackets of the function they are defined in. For variables defined outside of functions, they are “global” variables and can be used throughout the file.

## 3.12 Matrices

Matrices will be constructed by the user giving values for the rows and columns. A semicolon (`;`) will be used to distinguish between rows. Commas (`,`) will be used to distinguish between values in each row (differing columns). A matrix can only be of type `int` or `float`. For example:

```
matrix<int> m = [8, 1; 17, 4];
matrix<float> n = [8.0, 1.6; 17.34, 4.2];
```

Matrices cannot be returned in functions because they are being passed by reference.

### 3.12.1 Matrix Operations

1. Access: Indexing into an array is done using the [] brackets which take in two int expressions and returns the value accessed at the location specified by the expressions.

```
matrix<int> m = [1, 2; 3, 4; 5, 6];
int item = m[1,0];
/* item = 3 */
```

   This also allows for updating a specific location of the matrix at that index.

```
matrix<int> m = [1, 2; 3, 4; 5, 6];
m[1,0] = 8;
/* [1, 2; 8, 4; 5, 6]*/
```

2. Matrix Addition (+) and Subtraction (-): Adds and subtracts corresponding elements of matrices together. Both matrices must be of the same type and dimension.

```
matrix<int> m = [1, 2; 3, 4; 5, 6];
matrix<int> n = [6, 5; 4, 3; 2, 1];
matrix<int> add = m + n
/* [7, 7; 7, 7; 7, 7] */
matrix<int> sub = m - n
/* [-5, -3; -1, 1; 3, 5] */
```

3. Matrix Multiplication (*): Performs matrix multiplication of two matrices. Both matrices must be of the same type and the columns of the first matrix must be equal to the rows of the second matrix.

```
matrix<int> m = [1, 2; 3, 4; 5, 6];
matrix<int> n = [1, 2, 3; 4, 5, 6];
matrix<int> result = m * n;
/* [9, 12, 15; 19, 26, 33; 29, 40, 51] */
```

4. Scalar Multiplication (*): Multiplies each element of the matrix by a float or int scalar. The resulting matrix will be the same type as the input matrix.

```
matrix<int> m = [1, 2; 3, 4; 5, 6];
matrix<int> result = 2 * n;
/* [2, 4; 6, 8; 10, 12] */
```

5. Equals (==) and Not Equals (!=): Does an element by element equality check of each element in the matrix. Returns 1 if all elements are equal. Both matrices must be of the same type and dimension.

```
matrix<int> m = [1, 2; 3, 4; 5, 6];
matrix<int> n = [6, 5; 4, 3; 2, 1];
if (m == n) { print(1); }
else { print(0); }
/* will print 0 */
```

#### 3.12.2 Built-in Matrix Functions

1. `getRows`: Takes in a matrix of any type and returns an int depicting the number of rows within the matrix.

```
matrix<int> m = [1, 2; 3, 4; 5, 6];
int r = getRows(m);              /* r = 3 */
```

2. `getColumns`: Takes in a matrix of any type and returns an int depicting the number of columns within the matrix

```
matrix<int> m = [1, 2; 3, 4; 5, 6];
int c = getColumns(m);           /* c = 2 */
```

# 4 Project Plan

## 4.1 Process

Our team chose to meet at least once a week for brief check-ins. These meetings were stand-up style where everyone went around and discussed the progress they made prior to the meeting and their goals or the work they have moving into the next week. We would look at the work we had to do and try to split it up amongst group members and form sub-teams to tackle the tasks. Many times we would schedule other calls between the check-in meetings to debug code together or walk through implementation ideas.

After deciding on a matrix manipulation language, we looked through the MicroC language to decide what other programming basics we wanted to include in our language (loops, conditionals, types of variables). This allowed us to formalize the idea for our language in our LRM, though our language has continued to evolve throughout the entire process.

After using MicroC as our jumping off point, our team began to implement some MatrixMania specific functionality. We struggled through storing variables since we did not have global and local variables defined in the same way MicroC did but eventually figured it out. We also struggled with matrix implementation as we had to parse the matrix literal into a usable form that gave us information on its rows and column size. We went back and forth with edits in Semant and CodeGen to get these two problem areas working.

We made tests as we coded. As a team member added functionality to our language, they would write a test to ensure the functionality they added worked as expected. Google docs was used as a collaborative work space. We used it for meeting notes, idea planning, and drafts of turn-ins. We used a google calendar invite with a zoom link for our weekly meetings. These two tools allowed our team to stay connected and organized throughout the semester despite not being able to be physically present together.

## 4.2 Style Guide

Our team mostly used the MicroC style guide but highlighted a few important guidelines:

- Variable names: underscores between words

- 4-space indentation
- Matching alignment of list elements, aligning elements like |, {, }, in, and $->$ where applicable.
- test before a test program's name that is expected to pass
- fail before a test program's name that is expected to fail

## 4.3 Timeline

| Time | Task | Details |
|---|---|---|
| January 11 | Team Formation | We formed our team after the first lecture. |
| January 20-February 3 | Idea Generation | Group members had to come up with 3 indivual ideas before we met to vote on a final idea |
| February 3 | Project Proposal | Deliverable |
| February 14 | Development Environment | Setup of GitHub |
| February 16-February 22 | Development | Split into two teams. Initial grammar defined. Parser, Scanner, and LRM completed. |
| February 24 | LRM and Parser | Deliverable |
| March 10-March 24 | Development | Worked on Ast, Sast, Codegen, Semant for Hello World deliverable. Specifically, *print* function and compilation of first program in language (Hello World) |
| March 24 | Hello World | Deliverable |
| March 25-March 30 | Development | Codegen additional features added ($if/else$, $VarDecl$), test scripts, Semant updated to include full semantic checking |
| March 30-April 6 | Development | test scripts, Codegen fix for $elif$, Codegen additional features added (loops), switch to $Hashtbl$ in Codegen for $VarDecl$, Semant change for $VarDecl$ |
| April 7-April 12 | Development | Codegen addition of matrix counting of rows and columns, fix of Semant errors with matrix_expr, Semant issue with $VarDecl$ resolution |
| April 17 | Development and Testing | "Hack-a-thon" day to resolve outstanding issues with loops, finish matrix implementation in Codegen, additional tests written for matrices, final report writing |

| | | |
|---|---|---|
| April 17-25 | Testing and Preparation for Turn in | Final testing, slides made, LRM editing, and final report writing |
| April 26 | Final Project | Deliverable |

## 4.4 Roles and Responsibilities

| **Member** | **Original Assigned Role** |
|---|---|
| Sophie Reese-Wirpsa | Project Manager |
| Desu Imudia | Language Guru |
| Diego Prado | System Architect |
| Emily Ringel | System Architect |
| Cindy Espinosa | Tester |

## 4.5 Development Environment

Programming: OCaml v4.0.5
Code Generation/Optimization: LLVM v3.7
Version control: Git

Team members used IDEs of their choice to access and edit the code.

## 4.6 Project Log

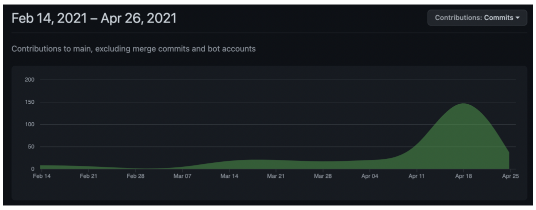



Figure 1: Commits from all team members across entire project

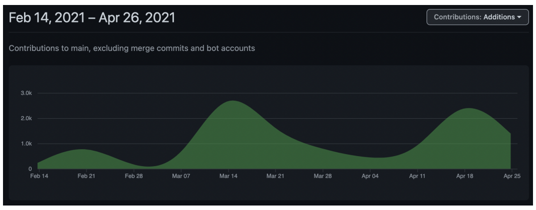



Figure 2: Additions from all team members across entire project

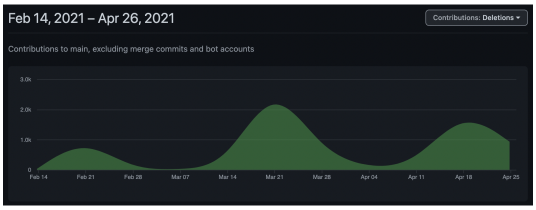
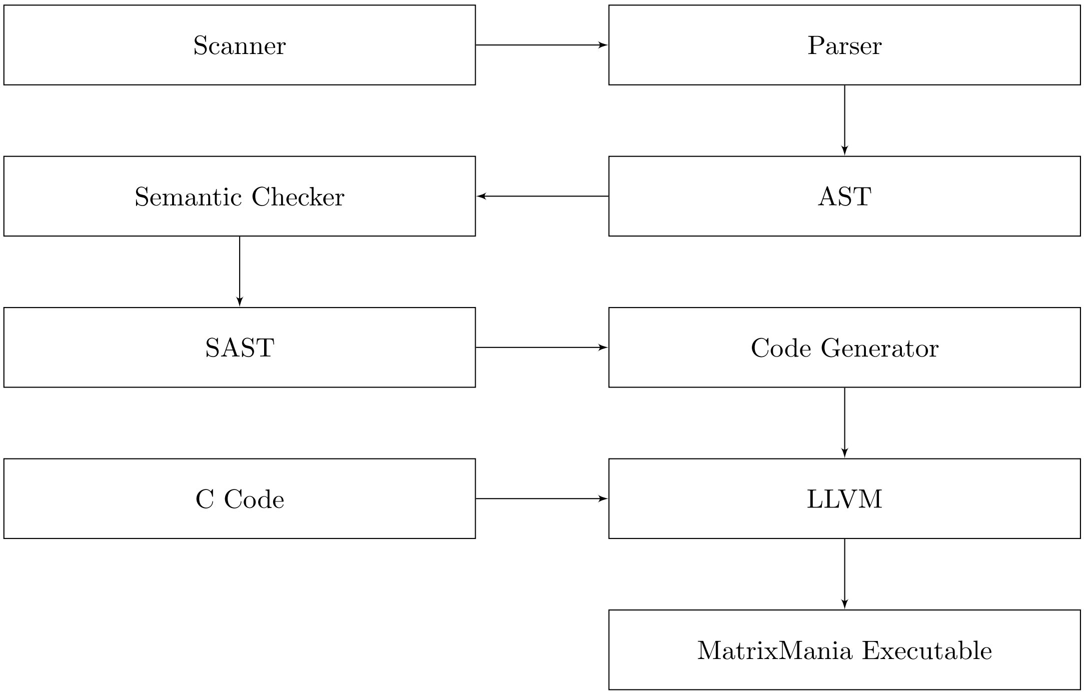



Figure 3: Deletions from all team members across entire project

# 5 Architectural Design

## 5.1 Compiler Diagram




## 5.2 Scanner

*Developed by Diego, Cindy*

`scanner.mll` is used to read in code. All of the white space and comments are disregarded and tokens are generated for anything with meaning. The scanner raises an exception if an illegal token is read in. The scanner outputs a stream of tokens using lexical analysis.

Our scanner is a standard scanner based off of MicroC's. We added in some tokens to allow for the manipulation of matrices and removed tokens that are not supported in our language. `elif`, `def`, and `matrix` are a few of the additions that are not found in MicroC's scanner. We removed `true`, `false`, and `bool` tokens as our language does not support boolean values.

### 5.3 Parser

*Developed by Diego, Cindy, Desu, Emily*

`parser.mly` receives a stream of tokens from the scanner and constructs an abstract syntax tree. To avoid constructing an ambiguous grammar, our team defined precedence and associativities in the parser. While our parser is very similar to the MicroC one, there are a few distinctions. We choose to implement `if`, `elif`, `else` control flow statements as opposed to just `if`, `else` as MicroC did. Our parser also turns `for` loops into `while` loops.

### 5.4 Semantic Checking

*Developed by Diego, Sophie, Desu, Emily*

`semant.ml` checks the AST generated by the parser to confirm that there are not any syntax rule violations. The types of issues caught during this part are not the same as in the parser (missing semi-colons or brackets), instead, semantic checking will confirm that what the user is asking the program to do is allowed. For example, assigning an int to a float is allowed but assigning a float to an int is not so semant would throw an error. Semant also checks which operators can work for each data type, and throws errors when they are applied to the wrong types.

Semant also constructs a table to store variable names and functions so that it can throw an exception if a user tries to access something that has not been declared. Because our variable declarations are defined as statements anywhere in a function rather than as local variables defined at the top of the function, we had to adapt this table from an immutable StringMap. We passed an environment variable holding a StringMap through each call to semantically check an expression or statement. Each time a new variable is defined, a new StringMap is created that includes that variable, which is then passed into the next statement.

### 5.5 Code Generation

*Developed by Diego, Emily, Sophie, Desu, Cindy*

`codegen.ml` generates LLVM IR code from the nodes in the SAST created by the semantic checker. During this process it checks that the llvm code that is generated is valid. The IR code is then turned into assembly code and an executable is created.

Similarly to semant, the table holding the pointers to each variable had to be adapted from the immutable StringMap used in MicroC. Here, we replaced the immutable StringMap with a mutable Hashtbl. That way, new variables are added to the existing Hashtbl as they arise.

Codegen also builds the LLVM representation of our matrix types. We wanted to make sure that matrices were easily accessible, not only from a syntax standpoint but also in terms of runtime and space complexity. MatrixMania internally holds matrices in one dimensional arrays, where the first two elements are the rows and columns. That way, index [i,j] of our matrix can be accessed in O(1) time by computing $(columns * i) + j + 2$.

Lastly, codegen interfaces with c in matrix_functions.c for all built-in functions. Some semantic checking is also included in our c functions to check the dimensions of the matrices passed in to the function, and will throw a run-time exception if there is an error.

# 6 Test Plan

## 6.1 Source Language Programs

Matrix Inverse:

Listing 3: tests/test_matrix_inverse.mm

```
/* Algorithms modified from https://www.geeksforgeeks.org/adjoint-inverse-matrix/
    */

/* Function to get cofactor of A[p,q] in temp.
   n is current dimension of A */
def void getCofactor(matrix<float> A , matrix<float> temp, int p, int q, int n ) {
    int i = 0;
    int j = 0;

    /* Looping for each element of the matrix */
    for (int row = 0; row < n; row = row + 1) {
        for (int col = 0; col < n; col = col + 1) {
            /*  Copying into temporary matrix only those element
                which are not in given row and column */
            if (row != p && col != q){
                temp[i, j] = A[row, col];
                j = j + 1;

                /*  Row is filled, so increase row index and
                    reset col index */
                if (j == n - 1) {
                    j = 0;
                    i = i+1;
                }
            }
        }
    }
}

/* Recursive function for finding determinant of matrix.
   n is current dimension of A. */
def float determinant(matrix<float> A, int n) {
    float D = 0;

    /*  Base case : if matrix contains single element */
    if (n == 1) {
        return A[0, 0];
    }

    matrix<float> tmp =
                    [0., 0., 0., 0.;
```

```
                    0., 0., 0., 0.;
                    0., 0., 0., 0.;
                    0., 0., 0., 0.];

    int sign = 1;  /* To store sign multiplier */

     /* Iterate for each element of first row */
    for (int f = 0; f < n; f=f+1) {
        /* Getting Cofactor of A[0, f] */
        getCofactor(A, tmp, 0, f, n);
        D = D + sign * A[0, f] * determinant(tmp, n - 1);

        /* terms are to be added with alternate sign */
        sign = -sign;
    }

    return D;
}

/* Function to get adjoint of A[N, N] in adj[N, N]. */
def void adjoint(matrix<float> A, matrix<float> adj) {
    int N = getRows(A);
    if (N == 1) {
        adj[0, 0] = 1;
        return;
    }
    /* temp is used to store cofactors of A */
    int sign = 1;
    matrix<float> tmp =
                   [0., 0., 0., 0.;
                    0., 0., 0., 0.;
                    0., 0., 0., 0.;
                    0., 0., 0., 0.];

    for (int i=0; i<N; i=i+1)
    {
        for (int j=0; j<N; j=j+1)
        {
            /* Get cofactor of A[i, j] */
            getCofactor(A, tmp, i, j, N);

            /* sign of adj[j, i] positive if sum of row
               and column indexes is even. */
            if ((i + j) % 2 == 0) {
                sign = 1;
            } else{
                sign = -1;
            }

            /* Interchanging rows and columns to get the
               transpose of the cofactor matrix */
            adj[j, i] = (sign)*(determinant(tmp, N-1));
        }
    }

}

/* Function to calculate and store inverse, returns false if
   matrix is singular */
def int inverse(matrix<float> A, matrix<float> inverse) {
    /* Find determinant of A */
    int N = getRows(A);
```

```
    float det = determinant(A, N);
    if (det == 0) {
        return 1;
    }

    /* Find adjoint */
    matrix<float> adj =
                    [0., 0., 0., 0.;
                     0., 0., 0., 0.;
                     0., 0., 0., 0.;
                     0., 0., 0., 0.];
    adjoint(A, adj);

    /* Find Inverse using formula "inverse(A) = adj(A)/det(A)" */
    for (int i=0; i<N; i=i+1) {
        for (int j=0; j<N; j=j+1) {
            inverse[i, j] = adj[i, j]/det;
        }
    }

    return 0;
}

def int main() {
    /* declare matrix */
    matrix<float> A = [1., 2., 3., 4.;
                       2., 5., 6., 7.;
                       3., 6., 8., 9.;
                       4., 7., 9., 10.];

    /* declare result matrix */
   matrix<float> A_inv =
                    [0., 0., 0., 0.;
                     0., 0., 0., 0.;
                     0., 0., 0., 0.;
                     0., 0., 0., 0.];

    /* compute invers/check if inverse is valid */
    if (inverse(A, A_inv) != 0) {
        print(-1);
    }
    printmf(A_inv);
    print(1111111111);
    printmf(A*A_inv);
}
```

```
; ModuleID = 'MatrixMania'
source_filename = "MatrixMania"

@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.1 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
@fmt.2 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.3 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
@fmt.4 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.5 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
@fmt.6 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.7 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
@fmt.8 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
```

```
@fmt.9 = private unnamed_addr constant [4 x i8] c"%g\0A\00"

declare i32 @printf(i8*, ...)

declare i32 @printm(i32*)

declare i32 @printmf(double*)

declare i32* @addm(i32*, i32*)

declare double* @addmf(double*, double*)

declare i32* @subm(i32*, i32*)

declare double* @submf(double*, double*)

declare i32* @scalarm(double, i32*)

declare double* @scalarmf(double, double*)

declare i32* @multiplication(i32*, i32*)

declare double* @multiplicationf(double*, double*)

declare i32 @equal(i32*, i32*)

declare i32 @equalf(double*, double*)

define i32 @main() {
entry:
  %A = alloca double*
  %matrix = alloca [18 x double]
  %ptr = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 0
  store double 4.000000e+00, double* %ptr
  %ptr1 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 1
  store double 4.000000e+00, double* %ptr1
  %ptr2 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 2
  store double 1.000000e+00, double* %ptr2
  %ptr3 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 3
  store double 2.000000e+00, double* %ptr3
  %ptr4 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 4
  store double 3.000000e+00, double* %ptr4
  %ptr5 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 5
  store double 4.000000e+00, double* %ptr5
  %ptr6 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 6
  store double 2.000000e+00, double* %ptr6
  %ptr7 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 7
  store double 5.000000e+00, double* %ptr7
  %ptr8 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 8
```

```
store double 6.000000e+00, double* %ptr8
%ptr9 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 9
store double 7.000000e+00, double* %ptr9
%ptr10 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 10
store double 3.000000e+00, double* %ptr10
%ptr11 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 11
store double 6.000000e+00, double* %ptr11
%ptr12 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 12
store double 8.000000e+00, double* %ptr12
%ptr13 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 13
store double 9.000000e+00, double* %ptr13
%ptr14 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 14
store double 4.000000e+00, double* %ptr14
%ptr15 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 15
store double 7.000000e+00, double* %ptr15
%ptr16 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 16
store double 9.000000e+00, double* %ptr16
%ptr17 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 17
store double 1.000000e+01, double* %ptr17
%matrix18 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 0
store double* %matrix18, double** %A
%A_inv = alloca double*
%matrix19 = alloca [18 x double]
%ptr20 = getelementptr inbounds [18 x double], [18 x double]* %matrix19, i64 0, i64 0
store double 4.000000e+00, double* %ptr20
%ptr21 = getelementptr inbounds [18 x double], [18 x double]* %matrix19, i64 0, i64 1
store double 4.000000e+00, double* %ptr21
%ptr22 = getelementptr inbounds [18 x double], [18 x double]* %matrix19, i64 0, i64 2
store double 0.000000e+00, double* %ptr22
%ptr23 = getelementptr inbounds [18 x double], [18 x double]* %matrix19, i64 0, i64 3
store double 0.000000e+00, double* %ptr23
%ptr24 = getelementptr inbounds [18 x double], [18 x double]* %matrix19, i64 0, i64 4
store double 0.000000e+00, double* %ptr24
%ptr25 = getelementptr inbounds [18 x double], [18 x double]* %matrix19, i64 0, i64 5
store double 0.000000e+00, double* %ptr25
%ptr26 = getelementptr inbounds [18 x double], [18 x double]* %matrix19, i64 0, i64 6
store double 0.000000e+00, double* %ptr26
%ptr27 = getelementptr inbounds [18 x double], [18 x double]* %matrix19, i64 0, i64 7
store double 0.000000e+00, double* %ptr27
%ptr28 = getelementptr inbounds [18 x double], [18 x double]* %matrix19, i64 0, i64 8
store double 0.000000e+00, double* %ptr28
%ptr29 = getelementptr inbounds [18 x double], [18 x double]* %matrix19, i64 0, i64 9
store double 0.000000e+00, double* %ptr29
%ptr30 = getelementptr inbounds [18 x double], [18 x double]* %matrix19, i64 0, i64 10
store double 0.000000e+00, double* %ptr30
%ptr31 = getelementptr inbounds [18 x double], [18 x double]* %matrix19, i64 0, i64 11
store double 0.000000e+00, double* %ptr31
%ptr32 = getelementptr inbounds [18 x double], [18 x double]* %matrix19, i64 0, i64 12
store double 0.000000e+00, double* %ptr32
```

```
  %ptr33 = getelementptr inbounds [18 x double], [18 x double]* %matrix19, i64 0, i64 13
  store double 0.000000e+00, double* %ptr33
  %ptr34 = getelementptr inbounds [18 x double], [18 x double]* %matrix19, i64 0, i64 14
  store double 0.000000e+00, double* %ptr34
  %ptr35 = getelementptr inbounds [18 x double], [18 x double]* %matrix19, i64 0, i64 15
  store double 0.000000e+00, double* %ptr35
  %ptr36 = getelementptr inbounds [18 x double], [18 x double]* %matrix19, i64 0, i64 16
  store double 0.000000e+00, double* %ptr36
  %ptr37 = getelementptr inbounds [18 x double], [18 x double]* %matrix19, i64 0, i64 17
  store double 0.000000e+00, double* %ptr37
  %matrix38 = getelementptr inbounds [18 x double], [18 x double]* %matrix19, i64 0, i64 0
  store double* %matrix38, double** %A_inv
  %A_inv39 = load double*, double** %A_inv
  %A40 = load double*, double** %A
  %inverse_result = call i32 @inverse(double* %A40, double* %A_inv39)
  %tmp = icmp ne i32 %inverse_result, 0
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
  %A_inv41 = load double*, double** %A_inv
  %printmf = call i32 @printmf(double* %A_inv41)
  %printf42 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt, i3
  %A43 = load double*, double** %A
  %A_inv44 = load double*, double** %A_inv
  %matmf = call double* @multiplicationf(double* %A_inv44, double* %A43)
  %printmf45 = call i32 @printmf(double* %matmf)
  ret i32 0

then:                                             ; preds = %entry
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt, i32
  br label %merge

else:                                             ; preds = %entry
  br label %merge
}

define i32 @inverse(double* %A, double* %inverse) {
entry:
  %A1 = alloca double*
  store double* %A, double** %A1
  %inverse2 = alloca double*
  store double* %inverse, double** %inverse2
  %N = alloca i32
  %A3 = load double*, double** %A1
  %rows = load double, double* %A3
  %rowsint = fptosi double %rows to i32
  store i32 %rowsint, i32* %N
  %det = alloca double
  %N4 = load i32, i32* %N
```

```
  %A5 = load double*, double** %A1
  %determinant_result = call double @determinant(double* %A5, i32 %N4)
  store double %determinant_result, double* %det
  %det6 = load double, double* %det
  %tmp = fcmp oeq double %det6, 0.000000e+00
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else
  %adj = alloca double*
  %matrix = alloca [18 x double]
  %ptr = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 0
  store double 4.000000e+00, double* %ptr
  %ptr7 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 1
  store double 4.000000e+00, double* %ptr7
  %ptr8 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 2
  store double 0.000000e+00, double* %ptr8
  %ptr9 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 3
  store double 0.000000e+00, double* %ptr9
  %ptr10 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 4
  store double 0.000000e+00, double* %ptr10
  %ptr11 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 5
  store double 0.000000e+00, double* %ptr11
  %ptr12 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 6
  store double 0.000000e+00, double* %ptr12
  %ptr13 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 7
  store double 0.000000e+00, double* %ptr13
  %ptr14 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 8
  store double 0.000000e+00, double* %ptr14
  %ptr15 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 9
  store double 0.000000e+00, double* %ptr15
  %ptr16 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 10
  store double 0.000000e+00, double* %ptr16
  %ptr17 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 11
  store double 0.000000e+00, double* %ptr17
  %ptr18 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 12
  store double 0.000000e+00, double* %ptr18
  %ptr19 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 13
  store double 0.000000e+00, double* %ptr19
  %ptr20 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 14
  store double 0.000000e+00, double* %ptr20
  %ptr21 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 15
  store double 0.000000e+00, double* %ptr21
  %ptr22 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 16
  store double 0.000000e+00, double* %ptr22
  %ptr23 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 17
  store double 0.000000e+00, double* %ptr23
  %matrix24 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 0
  store double* %matrix24, double** %adj
  %adj25 = load double*, double** %adj
```

```
  %A26 = load double*, double** %A1
  call void @adjoint(double* %A26, double* %adj25)
  %i = alloca i32
  store i32 0, i32* %i
  br label %while

then:                                             ; preds = %entry
  ret i32 1

else:                                             ; preds = %entry
  br label %merge

while:                                            ; preds = %merge51, %merge
  %i54 = load i32, i32* %i
  %N55 = load i32, i32* %N
  %tmp56 = icmp slt i32 %i54, %N55
  br i1 %tmp56, label %while_body, label %merge57

while_body:                                       ; preds = %while
  %j = alloca i32
  store i32 0, i32* %j
  br label %while27

while27:                                          ; preds = %while_body28, %while_body
  %j48 = load i32, i32* %j
  %N49 = load i32, i32* %N
  %tmp50 = icmp slt i32 %j48, %N49
  br i1 %tmp50, label %while_body28, label %merge51

while_body28:                                     ; preds = %while27
  %inverse29 = load double*, double** %inverse2
  %i30 = load i32, i32* %i
  %j31 = load i32, i32* %j
  %ptr32 = getelementptr inbounds double, double* %inverse29, i32 1
  %cols = load double, double* %ptr32
  %colsint = fptosi double %cols to i32
  %row = mul i32 %i30, %colsint
  %row_col = add i32 %row, %j31
  %idx = add i32 2, %row_col
  %ptr33 = getelementptr inbounds double, double* %inverse29, i32 %idx
  %adj34 = load double*, double** %adj
  %i35 = load i32, i32* %i
  %j36 = load i32, i32* %j
  %ptr37 = getelementptr inbounds double, double* %adj34, i32 1
  %cols38 = load double, double* %ptr37
  %colsint39 = fptosi double %cols38 to i32
  %row40 = mul i32 %i35, %colsint39
  %row_col41 = add i32 %row40, %j36
  %idx42 = add i32 2, %row_col41
```

```
  %ptr43 = getelementptr inbounds double, double* %adj34, i32 %idx42
  %element = load double, double* %ptr43
  %det44 = load double, double* %det
  %tmp45 = fdiv double %element, %det44
  store double %tmp45, double* %ptr33
  %j46 = load i32, i32* %j
  %tmp47 = add i32 %j46, 1
  store i32 %tmp47, i32* %j
  br label %while27

merge51:                                          ; preds = %while27
  %i52 = load i32, i32* %i
  %tmp53 = add i32 %i52, 1
  store i32 %tmp53, i32* %i
  br label %while

merge57:                                          ; preds = %while
  ret i32 0
}

define void @adjoint(double* %A, double* %adj) {
entry:
  %A1 = alloca double*
  store double* %A, double** %A1
  %adj2 = alloca double*
  store double* %adj, double** %adj2
  %N = alloca i32
  %A3 = load double*, double** %A1
  %rows = load double, double* %A3
  %rowsint = fptosi double %rows to i32
  store i32 %rowsint, i32* %N
  %N4 = load i32, i32* %N
  %tmp = icmp eq i32 %N4, 1
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else
  %sign = alloca i32
  store i32 1, i32* %sign
  %tmp7 = alloca double*
  %matrix = alloca [18 x double]
  %ptr8 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 0
  store double 4.000000e+00, double* %ptr8
  %ptr9 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 1
  store double 4.000000e+00, double* %ptr9
  %ptr10 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 2
  store double 0.000000e+00, double* %ptr10
  %ptr11 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 3
  store double 0.000000e+00, double* %ptr11
  %ptr12 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 4
```

```
  store double 0.000000e+00, double* %ptr12
  %ptr13 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 5
  store double 0.000000e+00, double* %ptr13
  %ptr14 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 6
  store double 0.000000e+00, double* %ptr14
  %ptr15 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 7
  store double 0.000000e+00, double* %ptr15
  %ptr16 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 8
  store double 0.000000e+00, double* %ptr16
  %ptr17 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 9
  store double 0.000000e+00, double* %ptr17
  %ptr18 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 10
  store double 0.000000e+00, double* %ptr18
  %ptr19 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 11
  store double 0.000000e+00, double* %ptr19
  %ptr20 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 12
  store double 0.000000e+00, double* %ptr20
  %ptr21 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 13
  store double 0.000000e+00, double* %ptr21
  %ptr22 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 14
  store double 0.000000e+00, double* %ptr22
  %ptr23 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 15
  store double 0.000000e+00, double* %ptr23
  %ptr24 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 16
  store double 0.000000e+00, double* %ptr24
  %ptr25 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 17
  store double 0.000000e+00, double* %ptr25
  %matrix26 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 0
  store double* %matrix26, double** %tmp7
  %i = alloca i32
  store i32 0, i32* %i
  br label %while

then:                                             ; preds = %entry
  %adj5 = load double*, double** %adj2
  %ptr = getelementptr inbounds double, double* %adj5, i32 1
  %cols = load double, double* %ptr
  %colsint = fptosi double %cols to i32
  %row = mul i32 0, %colsint
  %row_col = add i32 %row, 0
  %idx = add i32 2, %row_col
  %ptr6 = getelementptr inbounds double, double* %adj5, i32 %idx
  store double 1.000000e+00, double* %ptr6
  ret void

else:                                             ; preds = %entry
  br label %merge

while:                                            ; preds = %merge62, %merge
```

```
  %i65 = load i32, i32* %i
  %N66 = load i32, i32* %N
  %tmp67 = icmp slt i32 %i65, %N66
  br i1 %tmp67, label %while_body, label %merge68

while_body:                                       ; preds = %while
  %j = alloca i32
  store i32 0, i32* %j
  br label %while27

while27:                                          ; preds = %merge39, %while_body
  %j59 = load i32, i32* %j
  %N60 = load i32, i32* %N
  %tmp61 = icmp slt i32 %j59, %N60
  br i1 %tmp61, label %while_body28, label %merge62

while_body28:                                     ; preds = %while27
  %N29 = load i32, i32* %N
  %j30 = load i32, i32* %j
  %i31 = load i32, i32* %i
  %tmp32 = load double*, double** %tmp7
  %A33 = load double*, double** %A1
  call void @getCofactor(double* %A33, double* %tmp32, i32 %i31, i32 %j30, i32 %N29)
  %i34 = load i32, i32* %i
  %j35 = load i32, i32* %j
  %tmp36 = add i32 %i34, %j35
  %tmp37 = srem i32 %tmp36, 2
  %tmp38 = icmp eq i32 %tmp37, 0
  br i1 %tmp38, label %then40, label %else41

merge39:                                          ; preds = %else41, %then40
  %adj42 = load double*, double** %adj2
  %j43 = load i32, i32* %j
  %i44 = load i32, i32* %i
  %ptr45 = getelementptr inbounds double, double* %adj42, i32 1
  %cols46 = load double, double* %ptr45
  %colsint47 = fptosi double %cols46 to i32
  %row48 = mul i32 %j43, %colsint47
  %row_col49 = add i32 %row48, %i44
  %idx50 = add i32 2, %row_col49
  %ptr51 = getelementptr inbounds double, double* %adj42, i32 %idx50
  %sign52 = load i32, i32* %sign
  %N53 = load i32, i32* %N
  %tmp54 = sub i32 %N53, 1
  %tmp55 = load double*, double** %tmp7
  %determinant_result = call double @determinant(double* %tmp55, i32 %tmp54)
  %float_e1 = sitofp i32 %sign52 to double
  %tmp56 = fmul double %float_e1, %determinant_result
  store double %tmp56, double* %ptr51
```

```
  %j57 = load i32, i32* %j
  %tmp58 = add i32 %j57, 1
  store i32 %tmp58, i32* %j
  br label %while27

then40:                                           ; preds = %while_body28
  store i32 1, i32* %sign
  br label %merge39

else41:                                           ; preds = %while_body28
  store i32 -1, i32* %sign
  br label %merge39

merge62:                                          ; preds = %while27
  %i63 = load i32, i32* %i
  %tmp64 = add i32 %i63, 1
  store i32 %tmp64, i32* %i
  br label %while

merge68:                                          ; preds = %while
  ret void
}

define double @determinant(double* %A, i32 %n) {
entry:
  %A1 = alloca double*
  store double* %A, double** %A1
  %n2 = alloca i32
  store i32 %n, i32* %n2
  %D = alloca double
  store double 0.000000e+00, double* %D
  %n3 = load i32, i32* %n2
  %tmp = icmp eq i32 %n3, 1
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else
  %tmp6 = alloca double*
  %matrix = alloca [18 x double]
  %ptr7 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 0
  store double 4.000000e+00, double* %ptr7
  %ptr8 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 1
  store double 4.000000e+00, double* %ptr8
  %ptr9 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 2
  store double 0.000000e+00, double* %ptr9
  %ptr10 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 3
  store double 0.000000e+00, double* %ptr10
  %ptr11 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 4
  store double 0.000000e+00, double* %ptr11
  %ptr12 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 5
```

```
  store double 0.000000e+00, double* %ptr12
  %ptr13 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 6
  store double 0.000000e+00, double* %ptr13
  %ptr14 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 7
  store double 0.000000e+00, double* %ptr14
  %ptr15 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 8
  store double 0.000000e+00, double* %ptr15
  %ptr16 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 9
  store double 0.000000e+00, double* %ptr16
  %ptr17 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 10
  store double 0.000000e+00, double* %ptr17
  %ptr18 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 11
  store double 0.000000e+00, double* %ptr18
  %ptr19 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 12
  store double 0.000000e+00, double* %ptr19
  %ptr20 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 13
  store double 0.000000e+00, double* %ptr20
  %ptr21 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 14
  store double 0.000000e+00, double* %ptr21
  %ptr22 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 15
  store double 0.000000e+00, double* %ptr22
  %ptr23 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 16
  store double 0.000000e+00, double* %ptr23
  %ptr24 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 17
  store double 0.000000e+00, double* %ptr24
  %matrix25 = getelementptr inbounds [18 x double], [18 x double]* %matrix, i64 0, i64 0
  store double* %matrix25, double** %tmp6
  %sign = alloca i32
  store i32 1, i32* %sign
  %f = alloca i32
  store i32 0, i32* %f
  br label %while

then:                                             ; preds = %entry
  %A4 = load double*, double** %A1
  %ptr = getelementptr inbounds double, double* %A4, i32 1
  %cols = load double, double* %ptr
  %colsint = fptosi double %cols to i32
  %row = mul i32 0, %colsint
  %row_col = add i32 %row, 0
  %idx = add i32 2, %row_col
  %ptr5 = getelementptr inbounds double, double* %A4, i32 %idx
  %element = load double, double* %ptr5
  ret double %element

else:                                             ; preds = %entry
  br label %merge

while:                                            ; preds = %while_body, %merge
```

```
  %f52 = load i32, i32* %f
  %n53 = load i32, i32* %n2
  %tmp54 = icmp slt i32 %f52, %n53
  br i1 %tmp54, label %while_body, label %merge55

while_body:                                       ; preds = %while
  %n26 = load i32, i32* %n2
  %f27 = load i32, i32* %f
  %tmp28 = load double*, double** %tmp6
  %A29 = load double*, double** %A1
  call void @getCofactor(double* %A29, double* %tmp28, i32 0, i32 %f27, i32 %n26)
  %D30 = load double, double* %D
  %sign31 = load i32, i32* %sign
  %A32 = load double*, double** %A1
  %f33 = load i32, i32* %f
  %ptr34 = getelementptr inbounds double, double* %A32, i32 1
  %cols35 = load double, double* %ptr34
  %colsint36 = fptosi double %cols35 to i32
  %row37 = mul i32 0, %colsint36
  %row_col38 = add i32 %row37, %f33
  %idx39 = add i32 2, %row_col38
  %ptr40 = getelementptr inbounds double, double* %A32, i32 %idx39
  %element41 = load double, double* %ptr40
  %float_e1 = sitofp i32 %sign31 to double
  %tmp42 = fmul double %float_e1, %element41
  %n43 = load i32, i32* %n2
  %tmp44 = sub i32 %n43, 1
  %tmp45 = load double*, double** %tmp6
  %determinant_result = call double @determinant(double* %tmp45, i32 %tmp44)
  %tmp46 = fmul double %tmp42, %determinant_result
  %tmp47 = fadd double %D30, %tmp46
  store double %tmp47, double* %D
  %sign48 = load i32, i32* %sign
  %tmp49 = sub i32 0, %sign48
  store i32 %tmp49, i32* %sign
  %f50 = load i32, i32* %f
  %tmp51 = add i32 %f50, 1
  store i32 %tmp51, i32* %f
  br label %while

merge55:                                          ; preds = %while
  %D56 = load double, double* %D
  ret double %D56
}

define void @getCofactor(double* %A, double* %temp, i32 %p, i32 %q, i32 %n) {
entry:
  %A1 = alloca double*
  store double* %A, double** %A1
```

```
  %temp2 = alloca double*
  store double* %temp, double** %temp2
  %p3 = alloca i32
  store i32 %p, i32* %p3
  %q4 = alloca i32
  store i32 %q, i32* %q4
  %n5 = alloca i32
  store i32 %n, i32* %n5
  %i = alloca i32
  store i32 0, i32* %i
  %j = alloca i32
  store i32 0, i32* %j
  %row = alloca i32
  store i32 0, i32* %row
  br label %while

while:                                            ; preds = %merge45, %entry
  %row48 = load i32, i32* %row
  %n49 = load i32, i32* %n5
  %tmp50 = icmp slt i32 %row48, %n49
  br i1 %tmp50, label %while_body, label %merge51

while_body:                                       ; preds = %while
  %col = alloca i32
  store i32 0, i32* %col
  br label %while6

while6:                                           ; preds = %merge, %while_body
  %col42 = load i32, i32* %col
  %n43 = load i32, i32* %n5
  %tmp44 = icmp slt i32 %col42, %n43
  br i1 %tmp44, label %while_body7, label %merge45

while_body7:                                      ; preds = %while6
  %row8 = load i32, i32* %row
  %p9 = load i32, i32* %p3
  %tmp = icmp ne i32 %row8, %p9
  %col10 = load i32, i32* %col
  %q11 = load i32, i32* %q4
  %tmp12 = icmp ne i32 %col10, %q11
  %tmp13 = and i1 %tmp, %tmp12
  br i1 %tmp13, label %then, label %else39

merge:                                            ; preds = %else39, %merge35
  %col40 = load i32, i32* %col
  %tmp41 = add i32 %col40, 1
  store i32 %tmp41, i32* %col
  br label %while6
```

```
then:                                             ; preds = %while_body7
  %temp14 = load double*, double** %temp2
  %i15 = load i32, i32* %i
  %j16 = load i32, i32* %j
  %ptr = getelementptr inbounds double, double* %temp14, i32 1
  %cols = load double, double* %ptr
  %colsint = fptosi double %cols to i32
  %row17 = mul i32 %i15, %colsint
  %row_col = add i32 %row17, %j16
  %idx = add i32 2, %row_col
  %ptr18 = getelementptr inbounds double, double* %temp14, i32 %idx
  %A19 = load double*, double** %A1
  %row20 = load i32, i32* %row
  %col21 = load i32, i32* %col
  %ptr22 = getelementptr inbounds double, double* %A19, i32 1
  %cols23 = load double, double* %ptr22
  %colsint24 = fptosi double %cols23 to i32
  %row25 = mul i32 %row20, %colsint24
  %row_col26 = add i32 %row25, %col21
  %idx27 = add i32 2, %row_col26
  %ptr28 = getelementptr inbounds double, double* %A19, i32 %idx27
  %element = load double, double* %ptr28
  store double %element, double* %ptr18
  %j29 = load i32, i32* %j
  %tmp30 = add i32 %j29, 1
  store i32 %tmp30, i32* %j
  %j31 = load i32, i32* %j
  %n32 = load i32, i32* %n5
  %tmp33 = sub i32 %n32, 1
  %tmp34 = icmp eq i32 %j31, %tmp33
  br i1 %tmp34, label %then36, label %else

merge35:                                          ; preds = %else, %then36
  br label %merge

then36:                                           ; preds = %then
  store i32 0, i32* %j
  %i37 = load i32, i32* %i
  %tmp38 = add i32 %i37, 1
  store i32 %tmp38, i32* %i
  br label %merge35

else:                                             ; preds = %then
  br label %merge35

else39:                                           ; preds = %while_body7
  br label %merge

merge45:                                          ; preds = %while6
```

```
  %row46 = load i32, i32* %row
  %tmp47 = add i32 %row46, 1
  store i32 %tmp47, i32* %row
  br label %while

merge51:                                          ; preds = %while
  ret void
}
```

Matrix Transpose:

Listing 4: tests/test_transpose.mm

```
def void transpose(matrix<int> original, matrix<int> result){
    /* using built-in functions getRows and getColumns */
    int rows = getRows(original);
    int cols = getColumns(original);

    for(int i = 0; i < cols; i=i+1){
      for(int j = 0; j < rows; j=j+1){
            result[i, j] = original[j, i];
      }
    }
}

def int main(){
    /* declare matrix */
    matrix<int> A = [1, 2, 3;
                     4, 5, 6];
    printm(A);

    print(1111111111);

    /* declare result matrix */
    matrix<int> A_T = [0, 0;
                       0, 0;
                       0, 0];
    transpose(A, A_T);

    printm(A_T);

    return 0;
}
```

```
; ModuleID = 'MatrixMania'
source_filename = "MatrixMania"

@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.1 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
@fmt.2 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.3 = private unnamed_addr constant [4 x i8] c"%g\0A\00"

declare i32 @printf(i8*, ...)

declare i32 @printm(i32*)
```

```
declare i32 @printmf(double*)

declare i32* @addm(i32*, i32*)

declare double* @addmf(double*, double*)

declare i32* @subm(i32*, i32*)

declare double* @submf(double*, double*)

declare i32* @scalarm(double, i32*)

declare double* @scalarmf(double, double*)

declare i32* @multiplication(i32*, i32*)

declare double* @multiplicationf(double*, double*)

declare i32 @equal(i32*, i32*)

declare i32 @equalf(double*, double*)

define i32 @main() {
entry:
  %A = alloca i32*
  %matrix = alloca [8 x i32]
  %ptr = getelementptr inbounds [8 x i32], [8 x i32]* %matrix, i32 0, i32 0
  store i32 2, i32* %ptr
  %ptr1 = getelementptr inbounds [8 x i32], [8 x i32]* %matrix, i32 0, i32 1
  store i32 3, i32* %ptr1
  %ptr2 = getelementptr inbounds [8 x i32], [8 x i32]* %matrix, i32 0, i32 2
  store i32 1, i32* %ptr2
  %ptr3 = getelementptr inbounds [8 x i32], [8 x i32]* %matrix, i32 0, i32 3
  store i32 2, i32* %ptr3
  %ptr4 = getelementptr inbounds [8 x i32], [8 x i32]* %matrix, i32 0, i32 4
  store i32 3, i32* %ptr4
  %ptr5 = getelementptr inbounds [8 x i32], [8 x i32]* %matrix, i32 0, i32 5
  store i32 4, i32* %ptr5
  %ptr6 = getelementptr inbounds [8 x i32], [8 x i32]* %matrix, i32 0, i32 6
  store i32 5, i32* %ptr6
  %ptr7 = getelementptr inbounds [8 x i32], [8 x i32]* %matrix, i32 0, i32 7
  store i32 6, i32* %ptr7
  %matrix8 = getelementptr inbounds [8 x i32], [8 x i32]* %matrix, i32 0, i32 0
  store i32* %matrix8, i32** %A
  %A9 = load i32*, i32** %A
  %printm = call i32 @printm(i32* %A9)
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt, i32
  %A_T = alloca i32*
  %matrix10 = alloca [8 x i32]
```

```
  %ptr11 = getelementptr inbounds [8 x i32], [8 x i32]* %matrix10, i32 0, i32 0
  store i32 3, i32* %ptr11
  %ptr12 = getelementptr inbounds [8 x i32], [8 x i32]* %matrix10, i32 0, i32 1
  store i32 2, i32* %ptr12
  %ptr13 = getelementptr inbounds [8 x i32], [8 x i32]* %matrix10, i32 0, i32 2
  store i32 0, i32* %ptr13
  %ptr14 = getelementptr inbounds [8 x i32], [8 x i32]* %matrix10, i32 0, i32 3
  store i32 0, i32* %ptr14
  %ptr15 = getelementptr inbounds [8 x i32], [8 x i32]* %matrix10, i32 0, i32 4
  store i32 0, i32* %ptr15
  %ptr16 = getelementptr inbounds [8 x i32], [8 x i32]* %matrix10, i32 0, i32 5
  store i32 0, i32* %ptr16
  %ptr17 = getelementptr inbounds [8 x i32], [8 x i32]* %matrix10, i32 0, i32 6
  store i32 0, i32* %ptr17
  %ptr18 = getelementptr inbounds [8 x i32], [8 x i32]* %matrix10, i32 0, i32 7
  store i32 0, i32* %ptr18
  %matrix19 = getelementptr inbounds [8 x i32], [8 x i32]* %matrix10, i32 0, i32 0
  store i32* %matrix19, i32** %A_T
  %A_T20 = load i32*, i32** %A_T
  %A21 = load i32*, i32** %A
  call void @transpose(i32* %A21, i32* %A_T20)
  %A_T22 = load i32*, i32** %A_T
  %printm23 = call i32 @printm(i32* %A_T22)
  ret i32 0
}

define void @transpose(i32* %original, i32* %result) {
entry:
  %original1 = alloca i32*
  store i32* %original, i32** %original1
  %result2 = alloca i32*
  store i32* %result, i32** %result2
  %rows = alloca i32
  %original3 = load i32*, i32** %original1
  %rows4 = load i32, i32* %original3
  store i32 %rows4, i32* %rows
  %cols = alloca i32
  %original5 = load i32*, i32** %original1
  %ptr = getelementptr inbounds i32, i32* %original5, i32 1
  %cols6 = load i32, i32* %ptr
  store i32 %cols6, i32* %cols
  %i = alloca i32
  store i32 0, i32* %i
  br label %while

while:                                            ; preds = %merge, %entry
  %i30 = load i32, i32* %i
  %cols31 = load i32, i32* %cols
  %tmp32 = icmp slt i32 %i30, %cols31
```

```
  br i1 %tmp32, label %while_body, label %merge33

while_body:                                       ; preds = %while
  %j = alloca i32
  store i32 0, i32* %j
  br label %while7

while7:                                           ; preds = %while_body8, %while_body
  %j25 = load i32, i32* %j
  %rows26 = load i32, i32* %rows
  %tmp27 = icmp slt i32 %j25, %rows26
  br i1 %tmp27, label %while_body8, label %merge

while_body8:                                      ; preds = %while7
  %result9 = load i32*, i32** %result2
  %i10 = load i32, i32* %i
  %j11 = load i32, i32* %j
  %ptr12 = getelementptr inbounds i32, i32* %result9, i32 1
  %cols13 = load i32, i32* %ptr12
  %row = mul i32 %i10, %cols13
  %row_col = add i32 %row, %j11
  %idx = add i32 2, %row_col
  %ptr14 = getelementptr inbounds i32, i32* %result9, i32 %idx
  %original15 = load i32*, i32** %original1
  %j16 = load i32, i32* %j
  %i17 = load i32, i32* %i
  %ptr18 = getelementptr inbounds i32, i32* %original15, i32 1
  %cols19 = load i32, i32* %ptr18
  %row20 = mul i32 %j16, %cols19
  %row_col21 = add i32 %row20, %i17
  %idx22 = add i32 2, %row_col21
  %ptr23 = getelementptr inbounds i32, i32* %original15, i32 %idx22
  %element = load i32, i32* %ptr23
  store i32 %element, i32* %ptr14
  %j24 = load i32, i32* %j
  %tmp = add i32 %j24, 1
  store i32 %tmp, i32* %j
  br label %while7

merge:                                            ; preds = %while7
  %i28 = load i32, i32* %i
  %tmp29 = add i32 %i28, 1
  store i32 %tmp29, i32* %i
  br label %while

merge33:                                          ; preds = %while
  ret void
}
```

## 6.2 Test Suite

### 6.2.1 Passing Tests

Listing 5: tests/test_add.mm

```
def int add(int a, int b){
  return a + b;
}
def int main(){
    print(add(17,57));
    return 0;
}
```

Listing 6: outputs/test_add.out

```
74
```

Listing 7: tests/test_arith1.mm

```
def int main(){
    print(9+12);
}
```

Listing 8: outputs/test_arith1.out

```
21
```

Listing 9: tests/test_arith2.mm

```
def int main(){
    print(6*14*2*12);

}
```

Listing 10: outputs/test_arith2.out

```
2016
```

Listing 11: tests/test_arith3.mm

```
def int foo(int a) /*should be ignored*/
{
  return a;
}

def int main()
{
  int a = 3;
  a = a + 5;
  print(a);
}
```

Listing 12: outputs/test_arith3.out

```
8
```

Listing 13: tests/test_casting.mm

```
def int main(){
    int x = 0;
    float y = x;
    printf(y);
}
```

Listing 14: outputs/test_casting.out

```
0
```

Listing 15: tests/test_const_ops.mm

```
def int main()
{
  print(1 + 2);
  print(1 - 2);
  print(1 * 2);
  print(100 / 2);
  print(99);
  print(1 == 2);
  print(1 == 1);
  print(99);
  print(1 != 2);
  print(1 != 1);
  print(99);
  print(1 < 2);
  print(2 < 1);
  print(99);
  print(1 <= 2);
  print(1 <= 1);
  print(2 <= 1);
  print(99);
  print(1 > 2);
  print(2 > 1);
  print(99);
  print(1 >= 2);
  print(1 >= 1);
  print(2 >= 1);
  print(560%200);
}
```

Listing 16: outputs/test_const_ops.out

```
3
-1
2
50
99
0
1
99
1
0
99
1
0
99
1
1
0
```

```
99
0
1
99
0
1
1
160
```

Listing 17: tests/test_float_ops.mm

```
def 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);
}
def int main(){

    float c = 93.6;
    float d = 7.26;

    testfloat(c, d);

    testfloat(d, d);

}
```

Listing 18: outputs/test_float_ops.out

```
100.86
86.34
679.536
12.8926
0
1
1
0
1
1
0
0
14.52
0
52.7076
1
1
1
0
0
0
1
0
1
```

Listing 19: tests/test_float1.mm

```
def int main(){
  float a = 3.14159267;
  printf(a);
}
```

Listing 20: outputs/test_float1.out

```
3.14159
```

Listing 21: tests/test_float2.mm

```
def int main(){
    float x = 4.4567;
    float y = 8.913;
    float c = x + y;
    printf(c);
}
```

Listing 22: outputs/test_float2.out

```
13.3697
```

Listing 23: tests/test_for_loop1.mm

```
/* test simple for loop */
def int main(){
    for (int i = 0; i < 5; i=i+1){
        print(i);
    }
}
```

Listing 24: outputs/test_for_loop1.out

```
0
1
2
3
4
```

Listing 25: tests/test_for_loop2.mm

```
def int main(){
    for(int i = 4; i <= 8; i=i+1) {
    print(i);
  }
}
```

Listing 26: outputs/test_for_loop2.out

```
4
5
6
7
8
```

Listing 27: tests/test_func1.mm

```
def int add(int x, int z){
    return x + z;
}
```

```
def int main(){
    int y = add(7, 8);
    print(y);
}
```

Listing 28: outputs/test_func1.out

```
15
```

Listing 29: tests/test_function_call_param.mm

```
def int test(int z) {
    return z;
}

def int main(){
    int y = 4;
    int x = test(y);
    print(x);
}
```

Listing 30: outputs/test_function_call_param.out

```
4
```

Listing 31: tests/test_function_call.mm

```
def int test() {
    return 20;
}

def int main(){
    int x = test();
    print(x);
}
```

Listing 32: outputs/test_function_call.out

```
20
```

Listing 33: tests/test_gcd.mm

```
/*gcd program to showcase our language's ability to compute common cs problems */
def int gcd(int x, int y){
   if (x == 0){
      return y;
   }
   while(x != y){
      if(x > y) {x = x - y;}
      else {y = y - x;}
   }
   return x;
}
def int main( ){
   print(gcd(3, 15));
   print(gcd(18,24));
   print(gcd(45,120));
   return 0;
}
```

Listing 34: outputs/test_gcd.out

```
3
6
15
```

Listing 35: tests/test_if_elif_else_lit.mm

```
def int main(){
    if(0 == 1){
    print(0);
  }
  elif(1 == 1){
    print(1);
  }
  else {
    print(2);
  }
}
```

Listing 36: outputs/test_if_elif_else_lit.out

```
1
```

Listing 37: tests/test_if_elif_else_var.mm

```
def int main(){
 int x = 0;
 int y = 1;
 int z = 2;
 if(x == 0) {
     print(x);
 } elif(y == 1) {
     print(y);
 } else {
     print(z);
 }
}
```

Listing 38: outputs/test_if_elif_else_var.out

```
0
```

Listing 39: tests/test_if_elif_lit.mm

```
def int main(){
    if(0 == 0) {
        print(0);
    }
    elif (1 == 1) {
        print(1);
    }
}
```

Listing 40: outputs/test_if_elif_lit.out

```
0
```

Listing 41: tests/test_if_elif_var.mm

```
def int main(){
    int x = 1;
```

```
    if(0 == x) {
        print(0);
    }
    elif (1 == x) {
        print(1);
    }
}
```

Listing 42: outputs/test_if_elif_var.out

```
1
```

Listing 43: tests/test_int_float_ops.mm

```
/* operators on ints and floats */

def void testintfloat(int 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);
}
def int main(){

    int c = 80;
    float d = 5.5;

    testintfloat(c, d);
}
```

Listing 44: outputs/test_int_float_ops.out

```
85.5
74.5
440
14.5455
0
1
1
0
1
1
0
0
```

Listing 45: tests/test_int_ops.mm

```
/* operators on ints */

def void testint(int a, int b){
    print(a + b);
    print(a - b);
    print(a * b);
    print(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);
    print(a%b);
}
def int main(){

    int c = 80;
    int d = 5;

    testint(c, d);
}
```

Listing 46: outputs/test_int_ops.out

```
85
75
400
16
0
1
1
0
1
1
0
0
0
```

Listing 47: tests/test_mat_multiply.mm

```
def void multiply(matrix<int> x, matrix<int> y, matrix<int> empty){

  int x_rows = getRows(x);
  int y_cols = getColumns(y);
  int y_rows = getRows(y);

  for(int i = 0; i < x_rows; i=i+1){
    for(int j = 0; j < y_cols; j=j+1){
      for(int k = 0; k < y_rows; k =k+1) {
        empty[i,j] = empty[i,j] + x[i,k] * y[k,j];
      }
    }
  }
}

def int main(){

  matrix<int> m = [1, 0, 0;
                   0, 2, 0;
                   0, 0, 3];
  matrix<int> n = [1, 2, 3;
                   5, 6, 7;
                   8, 9, 10];
  matrix<int> empty3 = [0, 0, 0;
                        0, 0, 0;
                        0, 0, 0];
```

```
  multiply(m, n, empty3);
  printm(empty3);
  printm(m*n);
}
```

Listing 48: outputs/test_mat_multiply.out

```
1 2 3
10 12 14
24 27 30
1 4 9
5 12 21
8 18 30
```

Listing 49: tests/test_matrix_access.mm

```
def int main(){
  matrix<int> m = [1,4];
  print(m[0,0]);
}
```

Listing 50: outputs/test_matrix_access.out

```
1
```

Listing 51: tests/test_matrix_decl_float.mm

```
def int main(){
    matrix<float> x = [1.0];
    printmf(x);
}
```

Listing 52: outputs/test_matrix_decl_float.out

```
1.000000
```

Listing 53: tests/test_matrix_decl_int.mm

```
def int main(){
    matrix<int> x = [1];
    printm(x);
}
```

Listing 54: outputs/test_matrix_decl_int.out

```
1
```

Listing 55: tests/test_matrix_fl_ops.mm

```
def int main(){
    matrix<float> m1 = [.1,.2,.3;.4,.5,.6];
    matrix<float> m2 = [.6,.5,.4;.3,.2,.1];
    printmf(m1+m2);
    printmf(m1-m2);
    printmf(m1*3);
    printmf(3*m1);
    printmf(m1*3.5);
    printmf(3.5*m1);
    matrix<float> m3 = [1.1,2.2;3.3,4.4;5.5,6.6];
    printmf(m1*m3);
    print(m1==m3);
    print(m1!=m3);
}
```

Listing 56: outputs/test_matrix_fl_ops.out

```
0.700000 0.700000 0.700000
0.700000 0.700000 0.700000
-0.500000 -0.300000 -0.100000
0.100000 0.300000 0.500000
0.300000 0.600000 0.900000
1.200000 1.500000 1.800000
0.300000 0.600000 0.900000
1.200000 1.500000 1.800000
0.350000 0.700000 1.050000
1.400000 1.750000 2.100000
0.350000 0.700000 1.050000
1.400000 1.750000 2.100000
2.420000 3.080000
5.390000 7.040000
0
1
```

Listing 57: tests/test_matrix_int_ops.mm

```
def int main(){
    matrix<int> m1 = [1,2,3;4,5,6];
    matrix<int> m2 = [6,5,4;3,2,1];
    printm(m1+m2);
    printm(m1-m2);
    printm(m1*3);
    printm(3*m1);
    printm(m1*3.5);
    printm(3.5*m1);
    matrix<int> m3 = [1,2;3,4;5,6];
    printm(m1*m3);
    print(m1==m3);
    int x = (m1!=m3);
    print(x);
}
```

Listing 58: outputs/test_matrix_int_ops.out

```
7 7 7
7 7 7
-5 -3 -1
1 3 5
3 6 9
12 15 18
3 6 9
12 15 18
3 6 9
12 15 18
3 6 9
12 15 18
22 28
49 64
0
1
```

Listing 59: tests/test_matrix_mult_scal.mm

```
def void multiply_s(matrix<int> x, int y){

    int sizeOfR1 = getRows(x);
    int sizeOfC1= getColumns(x);
```

```

    for(int i = 0; i < sizeOfR1; i=i+1){
        for(int j = 0; j < sizeOfC1; j=j+1){
            x[i,j] = x[i,j] * y;
    }
    }
}


def int main(){

    matrix<int> m = [1,2,3;4,5,6];
    matrix<int> n = [1,2,3;4,5,6];
    int k = 2;

    printm(k * m);
    multiply_s(m, k);
    printm(m);
}
```

Listing 60: outputs/test_matrix_mult_scal.out

```
2 4 6
8 10 12
2 4 6
8 10 12
```

Listing 61: tests/test_matrix_pass_func.mm

```
def void pass(matrix<int> a){
  matrix<int> temp = a;
  printm(temp);
}
def int main(){
  matrix<int> a = [1,2];
  pass(a);
}
```

Listing 62: outputs/test_matrix_pass_func.out

```
1 2
```

Listing 63: tests/test_matrix_print.mm

```
def int main(){
  matrix<int> m = [1,3,5;2,4,6];
  printm(m);
}
```

Listing 64: outputs/test_matrix_print.out

```
1 3 5
2 4 6
```

Listing 65: tests/test_matrix_subtract.mm

```
def int main(){

    matrix<int> m = [1,2,3;4,5,6];
    matrix<int> n = [6,5,4;3,2,1];
    matrix<int> empty = [0,0,0;0,0,0];
```

```

    int sizeOfR1 = getRows(m);
    int sizeOfC1 = getColumns(m);

    for(int i = 0; i < sizeOfR1; i=i+1){
      for(int j = 0; j < sizeOfC1; j=j+1){
            empty[i,j] = m[i,j] - n[i,j];
        }
    }

    printm(empty);
    printm(m-n);

}
```

Listing 66: outputs/test_matrix_subtract.out

```
-5 -3 -1
1 3 5
-5 -3 -1
1 3 5
```

Listing 67: tests/test_return_test.mm

```
/* Semant Check -- return test */
def int retFive(){
    int x = 5;
    return x;
}

def int main(){
    int x = retFive();
    print(x);
}
```

Listing 68: outputs/test_return_test.out

```
5
```

Listing 69: tests/test_runtimeerr_mops1.mm

```
def int main(){
    matrix<int> m1 = [1,2,3;4,5,6];
    matrix<int> m2 = [6,5,4;3,2,1];
    printm(m1*m2);
}
```

Listing 70: outputs/test_runtimeerr_mops1.out

```
RUNTIME ERROR: matrices being multiplied do not have complementary dimensions.
```

Listing 71: tests/test_runtimeerr_mops2.mm

```
def int main(){
    matrix<int> m1 = [1,2,3;4,5,6];
    matrix<int> m3 = [1,2;3,4;5,6];
    printm(m1+m3);
}
```

Listing 72: outputs/test_runtimeerr_mops2.out

```
RUNTIME ERROR: matrices being added do not have the same dimensions.
```

Listing 73: tests/test_runtimeerr_mops3.mm

```
def int main(){
    matrix<float> m1 = [.1,.2,.3;.4,.5,.6];
    matrix<float> m2 = [.6,.5,.4;.3,.2,.1];
    printmf(m1*m2);
}
```

Listing 74: outputs/test_runtimeerr_mops3.out

```
RUNTIME ERROR: matrices being multiplied do not have complementary dimensions.
```

Listing 75: tests/test_runtimeerr_mops4.mm

```
def int main(){
    matrix<float> m1 = [.1,.2,.3;.4,.5,.6];
    matrix<float> m3 = [1.1,2.2;3.3,4.4;5.5,6.6];
    printmf(m1+m3);
}
```

Listing 76: outputs/test_runtimeerr_mops4.out

```
RUNTIME ERROR: matrices being added do not have the same dimensions.
```

Listing 77: tests/test_scope.mm

```
/* Semant Check -- scoping */
def float add(float x){
    float y = 5.0;
    float z = 0.0;
    while(y>0){
        z = x + y;
        y = y-1.0;
    }
    return z;
}
def int main(){
    printf(add(6.0));
}
```

Listing 78: outputs/test_scope.out

```
7
```

Listing 79: tests/test_var_assign_expr.mm

```
def int main(){
    int x = 0;
    int y = -3;
    print(x);
    x = x + y;
    print(x);
    x = x + 2;
    print(x);
}
```

Listing 80: outputs/test_var_assign_expr.out

```
0
-3
-1
```

Listing 81: tests/test_var_assign.mm

```
def int main(){
    int x = 0;
    print(x);
    x = 1;
    print(x);
    x = 2;
    print(x);
}
```

Listing 82: outputs/test_var_assign.out

```
0
1
2
```

Listing 83: tests/test_var_decl.mm

```
def int main( ){
    int x = 0;
    int y = x;
    print(y);
}
```

Listing 84: outputs/test_var_decl.out

```
0
```

Listing 85: tests/test_while_lit.mm

```
def int main( ){
    int x = 1;
    while(0 == 1) {
        print(1);
    }
    print(0);
}
```

Listing 86: outputs/test_while_lit.out

```
0
```

Listing 87: tests/test_while_loop.mm

```
def int main(){
    int i = 0;
  int sum = 0;
    while(i < 101){
    sum = sum + i;
    i =i+1;
    }
  print(sum);
}
```

Listing 88: outputs/test_while_loop.out

```
5050
```

Listing 89: tests/test_while_var_update.mm

```
def int main(){
    int x = 0;
    int y = 1;
    while(x == y) {
        print(1);
    }
    print(0);
}
```

Listing 90: outputs/test_while_var_update.out

```
0
```

Listing 91: tests/test_while_var.mm

```
def int main(){
    int x = 0;
    while(x == 1) {
  print(1);
 }
    print(0);
}
```

Listing 92: outputs/test_while_var.out

```
0
```

### 6.2.2 Failing Tests

Listing 93: tests/fail_add.mm

```
def int add(int a, int b){
  return a + b;
}
def int main(){
    print(add(17,57.2));
}
```

Listing 94: outputs/fail_add.err

```
Fatal error: exception Failure("illegal argument found float expected int in 57.2"
    )
```

Listing 95: tests/fail_assign1.mm

```
def int main(){
  int i = 33;
  int i = 15;
  float j = 2.0;
  float j = 3.14;
  int i = 2.5; /* Fail assigns a float to an int */

  print(i);
  print(j);
}
```

Listing 96: outputs/fail_assign1.err

```
Fatal error: exception Failure("Type not correct")
```

Listing 97: tests/fail_assign3.mm

```
def void testvoid(){
  int p = 1;
}
def int main(){
  int i = testvoid(); /* Fail: assigning a void to an int */
}
```

Listing 98: outputs/fail_assign3.err

```
Fatal error: exception Failure("Type not correct")
```

Listing 99: tests/fail_casting.mm

```
/* Semant Check -- float = int */
def int main(){
    float x = 2.45;
    int y = x; /* Should throw an error */
    print(y);
}
```

Listing 100: outputs/fail_casting.err

```
Fatal error: exception Failure("Type not correct")
```

Listing 101: tests/fail_const_ops.mm

```
def int main()
{
  print(1 + 2);
  print(1 - 2);
  print(1 * 2);
  print(100 / 2);
  print(99);
  print(1 == 2);
  print(1 = 1); /* Fail: unexpected type */
  print(99);
  print(1 != 2);
  print(1 != 1);
  print(99);
  print(1 < 2);
  print(2 < 1);
  print(99);
  print(1 <= 2);
  print(1 <= 1);
  print(2 <= 1);
  print(99);
  print(1 > 2);
  print(2 > 1);
  print(99);
  print(1 >= 2);
  print(1 >= 1);
  print(2 >= 1);
}
```

Listing 102: outputs/fail_const_ops.err

```
Fatal error: exception Parsing.Parse_error
```

Listing 103: tests/fail_float_ops.mm

```
def 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);
    print(a%b);
}
def int main(){

    float c = 93.6;
    float d = 7.26;

    testfloat(c, d);

    testfloat(d, d);

}
```

Listing 104: outputs/fail_float_ops.err

```
Fatal error: exception Failure("illegal binary operator float % float")
```

Listing 105: tests/fail_float1.mm

```
def int main(){
  -3.5 AND 1; /* Fail: Float with AND */
}
```

Listing 106: outputs/fail_float1.err

```
Fatal error: exception Parsing.Parse_error
```

Listing 107: tests/fail_float2.mm

```
def int main(){
    float x = 4.4567;
    float y = 8.913;
    float c = x + y;
    printm(c); /* Fail: printf is for floats printm is for matrices */
}
```

Listing 108: outputs/fail_float2.err

```
Fatal error: exception Failure("illegal argument found float expected matrix of
    type: int in c")
```

Listing 109: tests/fail_for_loop1.mm

```
/*failing test: for loop*/
def int main( ){
    for(int i=0; ) { /* incorrect syntax */
        print(i)
    }
}
```

Listing 110: outputs/fail_for_loop1.err

```
Fatal error: exception Parsing.Parse_error
```

Listing 111: tests/fail_for_loop2.mm

```
/*failing test: for loop*/
def int main( ){
    for(i=0; i < 7; i=i+1) { /* incorrect syntax: i is not defined*/
        print(i)
    }
}
```

Listing 112: outputs/fail_for_loop2.err

```
Fatal error: exception Parsing.Parse_error
```

Listing 113: tests/fail_func1.mm

```
def int foo() {}

def int bar() {}

def int baz() {}

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

def int main(){
  return 0;
}
```

Listing 114: outputs/fail_func1.err

```
Fatal error: exception Failure("duplicate function bar")
```

Listing 115: tests/fail_function_call_param.mm

```
def int test(z) { /* Fail: variable declaration expected */
    return z;
}

def int main(){
    int y = 4;
    int x = test(y);
    print(x);
}
```

Listing 116: outputs/fail_function_call_param.err

```
Fatal error: exception Parsing.Parse_error
```

Listing 117: tests/fail_function_call.mm

```
/* failing function call */
def int test() {
    return 20;
}
def int main(){
    int x = send(); /* no function send */
    print(x);
}
```

Listing 118: outputs/fail_function_call.err

```
Fatal error: exception Failure("unrecognized function send")
```

Listing 119: tests/fail_gcd.mm

```
def int gcd(int x, int y){
   if (x == 0){
      return y;
   }
   while(x != y){
      if(x > y) {x = x - y;}
      else {y = y - x;}
   }
   return x;
}
def int main( ){
   print(gcd(3, 15));
   print(gcd(18,24));
   print(gcd(45)); /* Fail: expecting 2 arguments */
}
~
```

Listing 120: outputs/fail_gcd.err

```
Fatal error: exception Failure("illegal character ~")
```

Listing 121: tests/fail_if_elif_else_lit.mm

```
def int main(){
 if(0) {
     print(0);
 } else if (1) {
     print(1);
 } else if (0) {
     print(2);
 } else {
     print(3);
 }
}
} /* Fail: too many closing braces */
```

Listing 122: outputs/fail_if_elif_else_lit.err

```
Fatal error: exception Parsing.Parse_error
```

Listing 123: tests/fail_if_elif_undef_var.mm

```
/*fail if elif*/
def int main( ){
 if(x) {
     print(0);
 } elif(p) {
     print(1);
 }
}
/* x and p are not defined */
```

Listing 124: outputs/fail_if_elif_undef_var.err

```
Fatal error: exception Failure("undeclared identifier p")
```

Listing 125: tests/fail_if_else_var.mm

```
def int main(){
 int x = 0;
 int y = 1;
 int z = 2;
 if(x) {
     print(x);
 }
 else if () {
     print(y);
 }
 else {
     print(z);
 }

 }
```

Listing 126: outputs/fail_if_else_var.err

```
Fatal error: exception Parsing.Parse_error
```

Listing 127: tests/fail_int_float_ops.mm

```
/* operators on ints and floats */

def void testintfloat(int a, float b){
    print(a + b);
    print(a - b);
    print(a * b);
    print(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);
}
def int main(){

    int c = 80;
    float d = 5.5;

    testintfloat(c, d);
}
```

Listing 128: outputs/fail_int_float_ops.err

```
Fatal error: exception Failure("illegal argument found float expected int in a + b
    ")
```

Listing 129: tests/fail_int_ops.mm

```
/* operators on ints */

def void testint(int a, int b){
    print(a + b);
    print(a - b);
    print(a * b);
    print(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);
}
def int main(){

    int c = 80;
    float d = 5; /* Fail: incompatible types */

    testint(c, d);

    testint(d, d);
}
```

Listing 130: outputs/fail_int_ops.err

```
Fatal error: exception Failure("illegal argument found float expected int in d")
```

Listing 131: tests/fail_matrix_access.mm

```
def int main(){
  matrix<int> m = [1,4];
  printm(m[3,3]);
}
```

Listing 132: outputs/fail_matrix_access.err

```
Fatal error: exception Failure("illegal argument found int expected matrix of type
    : int in m 3 3")
```

Listing 133: tests/fail_matrix_decl_float.mm

```
def int main( ){
    matrix<float> x = [1];
}
```

Listing 134: outputs/fail_matrix_decl_float.err

```
Fatal error: exception Failure("Type not correct")
```

Listing 135: tests/fail_matrix_decl_int.mm

```
def int main(){
    matrix<int> x = [1.0];
}
```

Listing 136: outputs/fail_matrix_decl_int.err

```
Fatal error: exception Failure("Type not correct")
```

Listing 137: tests/fail_matrix_decl.mm

```
def int main(){
  matrix< > x = [1];
}
```

Listing 138: outputs/fail_matrix_decl.err

```
Fatal error: exception Parsing.Parse_error
```

Listing 139: tests/fail_matrix_index.mm

```
def int main(){
        matrix<int> m = [1,2,3,4,5];
        m[0] = 10; /*not how to index*/
        printm(m);
}
```

Listing 140: outputs/fail_matrix_index.err

```
Fatal error: exception Parsing.Parse_error
```

Listing 141: tests/fail_matrix_print.mm

```
def int main(){
  matrix<int> m = [1,3,5,6,7];
  printmf(m); /*Fail: printmf is for matrix of floats */
}
```

Listing 142: outputs/fail_matrix_print.err

```
Fatal error: exception Failure("illegal argument found matrix of type: int
    expected matrix of type: float in m")
```

Listing 143: tests/fail_return_scope.mm

```
/* fail because of return */
def int main(){
    int y = 9;
    return y;
    print(y); /*cannot print after a return*/
}
```

Listing 144: outputs/fail_return_scope.err

```
Fatal error: exception Failure("nothing may follow a return")
```

Listing 145: tests/fail_scope.mm

```
/* Semant Check -- scoping */
def float add(float x){
    float y = 5.0;
    while(y>1.0){
        float z = x + y;
    }
    return z;
}

def int main(){
    int x = 0;
    int y = x;
    printf(add(6.0));
}
```

Listing 146: outputs/fail_scope.err

```
Fatal error: exception Failure("undeclared identifier z")
```

Listing 147: tests/fail_var_assign.mm

```
/*failing test for assigning floats to an int var*/

def int main( ){
    float x = 1.0;
    int x = 5.5;
    print(x);
    print(y);
}
```

Listing 148: outputs/fail_var_assign.err

```
Fatal error: exception Failure("Type not correct")
```

Listing 149: tests/fail_var_no_type.mm

```
/*fail test for variable declaration*/
def int main( ){
   y = 1; /* type needed */

}
```

Listing 150: outputs/fail_var_no_type.err

```
Fatal error: exception Failure("undeclared identifier y")
```

Listing 151: tests/fail_while_condition.mm

```
/*failing test for while*/
def int main( ){
    while(x){ /*not a condition*/
        int x = 0;
    }

}
```

Listing 152: outputs/fail_while_condition.err

```
Fatal error: exception Failure("undeclared identifier x")
```

Listing 153: tests/fail_while_lit.mm

```
def int main(){
    int x = 1;
    while(0 = 1) { /* Fail: expecting a variable due to assignment operator */
        print(1);
    }
    print(0);
}
```

Listing 154: outputs/fail_while_lit.err

```
Fatal error: exception Parsing.Parse_error
```

## 6.3 Explanation

Our test suite is located in the tests directory and contains three different program extensions: .mm for programs written in our MatrixMania language, .out for expected output, and .err for expected error messages of invalid MatrixMania programs.

We implemented several different types of tests in our test suite for MatrixMania features below:

1. Types
   - ints
   - floats
2. Variables
   - Assignment
   - Scope
3. Control Flow & Loops
   - if, elif, else
   - while
   - for
4. Matrices
   - Arithmetic
   - Access
   - Assignment
   - Indexing

We also have our demo files and their expected output located in the test folder. Though these are not specific tests, we can still use them to test how the different pieces all fit together.

## 6.4 Reasoning

We tested throughout all stages of development. Before we had our full compiler working, we would debug through the OCaml interpreter. After the "Hello World" milestone and development on codegen, we were able to begin to write test cases in our MatrixMania language.

There are different types of tests to test each module we developed. For example, there are specific syntax errors that should result in the Parser throwing an error. This was written into tests so that we made such each of the different components worked and threw errors when necessary.

As we began to implement parts of our language, we would write a test case for it. For example, after `if`, `elif`, `else` were implemented in codegen, tests were written to ensure they worked with variables and constants.

## 6.5 Test Automation

We adapted the testall shell script from MicroC to run regression testing on all code in the test directory. To run the script a user must run `make all` then `./testall.sh`. Running the script produces an OK output if the tests passed or a FAILED output with an explanation as to why it failed.

We have another shell script called matrixrun.sh that can be used to run all the matrix tests and demo programs in the matrix_code directory. We decided to separate these specific programs out from the test directory since they were not specific to one singular feature of our language.

Instead, these combine different feature to do matrix manipulation such as transpose or inverse. We still wanted to be able to test these as other parts of our language was updated so they have .out files that they are compared to when the shell script is run.

We designed two types of tests: passing tests and failing tests. The output of a passing test is compared to the expected output which is located in a .out file with the same name as the test. The output of a failing test is an error. The error produced by running the code is compared to the expected error in the .err file. If the .out or .err file match the output, the test past otherwise they fail.

The shell script that is used to run all of the tests in our program:

Listing 155: testall.sh

```
#!/bin/sh

# Regression testing script for MicroC
# Step through a list of files
#  Compile, run, and check the output of each expected-to-work test
#  Compile and check the error of each expected-to-fail test

# Path to the LLVM interpreter
LLI="lli"
#LLI="/usr/local/opt/llvm/bin/lli"

# Path to the LLVM compiler
LLC="llc"

# Path to the C compiler
GCC="gcc"

# Path to the microc compiler.  Usually "./microc.native"
# Try "_build/microc.native" if ocamlbuild was unable to create a symbolic link.
MICROC="_build/matrixmania.native"
#MICROC="_build/microc.native"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.mm files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
  echo "FAILED"
  # error=1
    fi
    echo "  $1"
}

```

```
# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
  SignalError "$1 differs"
  echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
  SignalError "$1 failed on $*"
  return 1
    }
}

# RunFail <args>
# Report the command, run it, and expect an error
RunFail() {
    echo $* 1>&2
    eval $* && {
  SignalError "failed: $* did not report an error"
  return 1
    }
    return 0
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.mm//'`
    reffile=`echo $1 | sed 's/.mm$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.s ${basename}.exe ${basename}.out" &&
    Run "$MICROC" "$1" ">" "${basename}.ll" &&
    Run "$LLC" "-relocation-model=pic" "${basename}.ll" ">" "${basename}.s" &&
    Run "$GCC" "-o" "${basename}.exe" "${basename}.s" "c_functions/matrix_functions.o" &&
    Run "./${basename}.exe" > "${basename}.out" &&
    Compare ${basename}.out ${reffile}.out ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
  if [ $keep -eq 0 ] ; then
      rm -f $generatedfiles
  fi
  echo "OK"
```

```
  echo "###### SUCCESS" 1>&2
    else
  echo "###### FAILED" 1>&2
  globalerror=$error
    fi
}

CheckFail() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.mm//'`
    reffile=`echo $1 | sed 's/.mm$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
    RunFail "$MICROC" "<" $1 "2>" "${basename}.err" ">>" $globallog &&
    Compare ${basename}.err ${reffile}.err ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
  if [ $keep -eq 0 ] ; then
      rm -f $generatedfiles
  fi
  echo "OK"
  echo "###### SUCCESS" 1>&2
    else
  echo "###### FAILED" 1>&2
  globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
  k) # Keep intermediate files
      keep=1
      ;;
  h) # Help
      Usage
      ;;
    esac
done

shift `expr $OPTIND - 1`

LLIFail() {
  echo "Could not find the LLVM interpreter \"$LLI\"."
  echo "Check your LLVM installation and/or modify the LLI variable in testall.sh"
  exit 1
}

which "$LLI" >> $globallog || LLIFail

if [ $# -ge 1 ]
then
```

```
    files=$@
else
    files="tests/test_*.mm tests/fail_*.mm"
fi

for file in $files
do
    case $file in
  *test_*)
      Check $file 2>> $globallog
      ;;
  *fail_*)
      CheckFail $file 2>> $globallog
      ;;
  *)
      echo "unknown file type $file"
      globalerror=1
      ;;
    esac
done

exit $globalerror
```

The shell script that is used to run matrix programs and feature combination tests:

Listing 156: matrixrun.sh

```
#!/bin/sh

# Regression testing script for MicroC
# Step through a list of files
#  Compile, run, and check the output of each expected-to-work test
#  Compile and check the error of each expected-to-fail test

# Path to the LLVM interpreter
LLI="lli"
#LLI="/usr/local/opt/llvm/bin/lli"

# Path to the LLVM compiler
LLC="llc"

# Path to the C compiler
GCC="gcc"

# Path to the microc compiler.  Usually "./microc.native"
# Try "_build/microc.native" if ocamlbuild was unable to create a symbolic link.
MICROC="_build/matrixmania.native"
#MICROC="_build/microc.native"

# Set time limit for all operations
ulimit -t 30

globallog=matrixrun.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: matrixrun.sh [options] [.mm files]"
    echo "-k    Keep intermediate files"
```

```
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
  echo "FAILED"
  # error=1
    fi
    echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
  SignalError "$1 differs"
  echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
  SignalError "$1 failed on $*"
  return 1
    }
}

# RunFail <args>
# Report the command, run it, and expect an error
RunFail() {
    echo $* 1>&2
    eval $* && {
  SignalError "failed: $* did not report an error"
  return 1
    }
    return 0
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.mm//'`
    reffile=`echo $1 | sed 's/.mm$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.s ${basename}.exe ${basename}.out" &&
    Run "$MICROC" "$1" ">" "${basename}.ll" &&
    Run "$LLC" "-relocation-model=pic" "${basename}.ll" ">" "${basename}.s" &&
```

```
    Run "$GCC" "-o" "${basename}.exe" "${basename}.s" "c_functions/
    matrix_functions.o" &&
    Run "./${basename}.exe" > "${basename}.out" &&
    Compare ${basename}.out ${reffile}.out ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
  if [ $keep -eq 0 ] ; then
      rm -f $generatedfiles
  fi
  echo "OK"
  echo "###### SUCCESS" 1>&2
    else
  echo "###### FAILED" 1>&2
  globalerror=$error
    fi
}


while getopts kdpsh c; do
    case $c in
  k) # Keep intermediate files
      keep=1
      ;;
  h) # Help
      Usage
      ;;
    esac
done

shift `expr $OPTIND - 1`

LLIFail() {
  echo "Could not find the LLVM interpreter \"$LLI\"."
  echo "Check your LLVM installation and/or modify the LLI variable in testall.sh"
  exit 1
}

which "$LLI" >> $globallog || LLIFail

if [ $# -ge 1 ]
then
    files=$@
else
    files="matrix_tests/test_*.mm matrix_tests/fail_*.mm"
fi

for file in $files
do
    case $file in
  *test_*)
      Check $file 2>> $globallog
      ;;
  *fail_*)
      CheckFail $file 2>> $globallog
      ;;
  *)
      echo "unknown file type $file"
      globalerror=1
      ;;
    esac
```

```
done

exit $globalerror
```

### 6.6 Roles and Responsibilities

While Cindy was the tester, the test suite was a team wide effort. As codegen was being developed, tests were written to check if the feature being added worked as expected. For example as Diego worked on `if`, `elif`, `else` statements in codegen, he wrote a test to check if they worked with constants, literals, and then wrote tests to check that they fail where we expected them to. Emily, Cindy, and Sophie adapted the shell scripts from MicroC to work with our modules and tests. Sophie edited many of the tests from MicroC and made .err and .outs with their expected outputs. Emily made matrix specific tests as she was working on addition, subtraction, and multiplication for matrices.

## 7 Lessons Learned

### 7.1 Advice

To future teams,

We recommend that you establish weekly meetings and an open line of communication through a group message of some form. The ability to quickly reach other team members played a pivotal role in our teams success. Another aspect of communication that was helpful for our team was understanding what else other members had going on that week. We would share openly about other exams or deadlines and be realistic about how much work we would be able to get done.

On a more technical note, we recommend understanding what each module does from the very beginning. While this may seem daunting as it will only be a few weeks into the course and you may have no idea what a code generator does, getting a basic understanding of the pipeline between all of the different components is critical for each members ability to contribute to the project meaningfully.

-MatrixMania Team

### 7.2 Individual Reflection

#### 7.2.1 Cindy Espinosa

My biggest takeaway was learning functional programming with OCaml. I learned about the compiler pipeline and how it all fits together. My advice for future students is to create your own deadlines in order to stay on track. I also learned about working with a software engineering group and how important communication is especially during remote times when we're not all able to meet together or at the same time. We would have "hackathon" type sessions that were super helpful in everyone being on the same page.

### 7.2.2 Desu Imudia

You may have all the ideas in the world but not be able to implement them all. There were a lot of features and components of our language that were ultimately scrapped due to lack of time, capability, and/or feasibility. This is where planning and communication become so incredibly important. As the language guru, it's best to start off with an image of what you want the language to look like, and continue to make known the little details. There were tough decisions to be made in terms of what would be possible to do or not. It's important in these moments to fully communicate what we were struggling with and to ask for help. I'm grateful that I was with a group that trusted each other enough to be transparent at every turn. I also learned so much about coding in general. The learning curve was huge, but even knowing little tidbits here and there helped in the long run.

### 7.2.3 Diego Prado

Working with a team, having early, often, and clear deadlines are really important to making sure the project actually gets finished (even if the goals aren't met on time). I also realized the importance of having well documented and easy to read code early on and not just fixing it all up at the end. Because of all of the changes that we made to our language and our system throughout the project, it was handy to be able to go back and know exactly where a change needed to be made or where an issue was coming from. Similarly, I learned that really understanding how all of the different parts of the project are supposed to work together, especially early on, is really important in terms of knowing how to even start working on a feature.

### 7.2.4 Sophie Reese-Wirpsa

Understanding the language and different aspects of developments are key to success in this project, as I found out throughout this process. Since all of the modules are so closely linked, I found out that one must understand what all of them do even if you are just working on one subsection of one of the components. I also learned that having a well defined language with specific implementation goals at the beginning is very helpful so that group members do not have to question features during development. Being understanding of team member's outside responsibilities to set realistic goals also was helpful so that when we came to meetings we did not expect parts of the project to be complete when that was not possible some weeks.

### 7.2.5 Emily Ringel

Communication and understanding what your team members are working on are key to completing a project this large. You might not understand every line of code that you didn't write, but knowing where to look and who to ask when components have to be integrated is really important for completing each deliverable on time. Communication is also important when there are so many moving parts to make sure everything you need implemented works cohesively in each section of the project, from the parser to the code generator to the test suite. Lastly, I learned to manage my time wisely and take breaks when needed, especially when working on larger parts of the project. Many of the issues that come up require you to look at a problem from a new perspective, which often can't be fixed without a clear mind and some time away.

# 8 Appendix

## 8.1 AST

Listing 157: mods/ast.ml

```
(* ast.ml file *)
(*Emily, Diego, Sophie, Desu*)

(* Abstract Syntax Tree and functions for printing it *)

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

type uop = Not | Neg

type typ = Int | Float | Void | Matrix of typ

type bind = typ * string

type expr =
    IntLit of int
  | FLit of string
  | MatrixLit of (expr list) list
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of string * expr
  | Call of string * expr list
  | Access of expr * expr * expr
  | Noexpr

type stmt =
    Block of stmt list
  | VarDecl of typ * string * expr
  | Update of expr * expr * expr * expr
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | While of expr * stmt

type func_decl = {
    typ : typ;
    fname : string;
    formals : bind list;
    body : stmt list;
  }

  type var_decl = {
    var: bind
  }

type program = func_decl list

(* Pretty-printing functions *)

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
```

```
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Mod -> "%"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"

(*Added - for pretty printing*)
let rec string_of_typ (t) =
  match t with
    Int -> "int"
  | Float -> "float"
  | Void -> "void"
  | Matrix(t) -> "matrix of type: " ^ string_of_typ t

let string_of_uop (o) =
  match o with
    Not -> "!"
  | Neg -> "-"

let rec string_of_expr = function
  IntLit(l) -> string_of_int l
| FLit(l) -> l
| Id(s) -> s
| 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 -> ""
| MatrixLit(l) ->
let string_of_row l =
  String.concat "" (List.map string_of_expr l)
in
String.concat "" (List.map string_of_row l)
| Access(e1, e2, e3) ->
string_of_expr e1 ^ " " ^ string_of_expr e2 ^ " " ^ string_of_expr e3

let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^ ";\n"

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, s1, s3) ->
    "if (" ^ string_of_expr e ^ ")\n" ^
    string_of_stmt s1 ^ "else\n" ^ string_of_stmt s3
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
  | VarDecl(t, s, e) -> string_of_typ t ^ " " ^ s ^ "=" ^ string_of_expr e ^ "\n"
  | Update(m, r, c, e) -> string_of_expr e ^ "[" ^ string_of_expr e ^ "," ^
                          string_of_expr e ^ "] =" ^ string_of_expr e ^ "\n"

let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map snd fdecl.formals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
```

```
  "}\n"
let string_of_program funcs =  String.concat "" (List.map string_of_fdecl funcs)
```

## 8.2 Code Generator

Listing 158: mods/codegen.ml

```
(*Emily, Diego, Sophie, Desu, Cindy*)
module L = Llvm
module A = Ast

open Sast

module StringMap = Map.Make(String)

let translate (functions) =
  let context = L.global_context () in
  let the_module = L.create_module context "MatrixMania" in

  let i32_t = L.i32_type context
  and i8_t = L.i8_type context
  and float_t    = L.double_type context
  and i64_t = L.i64_type context
  and void_t     = L.void_type   context in

  let rec ltype_of_typ = function
      A.Int -> i32_t
    | A.Float -> float_t
    | A.Void  -> void_t
    | A.Matrix(t) -> L.pointer_type (ltype_of_typ t)

  in


  (* functions to easily get number of rows/columns of a matrix *)
  let get_matrix_rows matrix builder = (* matrix has already gone through expr *)
    let typ = L.string_of_lltype (L.type_of matrix) in
    let ret = match typ with
      "double*" -> let rows = L.build_load matrix "rows" builder
        in L.build_fptosi rows i32_t "rowsint" builder
      | _ -> L.build_load matrix "rows" builder
    in ret
  in
  let get_matrix_cols matrix builder =
    let ptr = L.build_in_bounds_gep matrix [| L.const_int i32_t 1|] "ptr" builder
    in
    let typ = L.string_of_lltype (L.type_of ptr) in
    let ret = match typ with
      "double*" -> let cols = L.build_load ptr "cols" builder
        in L.build_fptosi cols i32_t "colsint" builder
      | _ -> L.build_load ptr "cols" builder
    in ret
  in

  (* Declare external functions *)

  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 printm_t : L.lltype =
    L.function_type i32_t [| L.pointer_type i32_t |] in
let printm_func : L.llvalue =
    L.declare_function "printm" printm_t the_module in

let printmf_t : L.lltype =
    L.function_type i32_t [| L.pointer_type float_t |] in
let printmf_func : L.llvalue =
    L.declare_function "printmf" printmf_t the_module in

(* Define each function (arguments and return type)
so we can call it even before we've created its body *)

  (* addition*)

  let addm_t : L.lltype =
    L.function_type (L.pointer_type i32_t) [| L.pointer_type i32_t; L.pointer_type i32_t |] in
let addm_func : L.llvalue =
    L.declare_function "addm" addm_t the_module in

let addmf_t : L.lltype =
    L.function_type (L.pointer_type float_t) [| L.pointer_type float_t; L.pointer_type float_t |] in
let addmf_func : L.llvalue =
    L.declare_function "addmf" addmf_t the_module in

(* subtraction *)
let subm_t : L.lltype =
    L.function_type (L.pointer_type i32_t) [| L.pointer_type i32_t; L.pointer_type i32_t |] in
let subm_func : L.llvalue =
    L.declare_function "subm" subm_t the_module in

let submf_t : L.lltype =
    L.function_type (L.pointer_type float_t) [| L.pointer_type float_t; L.pointer_type float_t |] in
let submf_func : L.llvalue =
    L.declare_function "submf" submf_t the_module in

(* scalar multiplication*)

let scalarm_t : L.lltype =
    L.function_type (L.pointer_type i32_t) [| float_t; L.pointer_type i32_t|] in
let scalarm_func : L.llvalue =
    L.declare_function "scalarm" scalarm_t the_module in

let scalarmf_t : L.lltype =
    L.function_type (L.pointer_type float_t) [| float_t; L.pointer_type float_t |] in
let scalarmf_func : L.llvalue =
    L.declare_function "scalarmf" scalarmf_t the_module in

(* matrix multiplication*)

let multiplication_t : L.lltype =
    L.function_type (L.pointer_type i32_t) [| L.pointer_type i32_t; L.pointer_type i32_t|] in
let multiplication_func : L.llvalue =
    L.declare_function "multiplication" multiplication_t the_module in

let multiplicationf_t : L.lltype =
```

```
    L.function_type (L.pointer_type float_t) [| L.pointer_type float_t; L.
  pointer_type float_t|] in
let multiplicationf_func : L.llvalue =
    L.declare_function "multiplicationf" multiplicationf_t the_module in

(* equals *)
let equal_t : L.lltype =
    L.function_type i32_t [| L.pointer_type i32_t; L.pointer_type i32_t|] in
let equal_func : L.llvalue =
    L.declare_function "equal" equal_t the_module in

let equalf_t : L.lltype =
    L.function_type i32_t [| L.pointer_type float_t; L.pointer_type float_t|] in
let equalf_func : L.llvalue =
    L.declare_function "equalf" equalf_t the_module in

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 functions 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
  and float_format_str = L.build_global_stringptr "%g\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 var_hash = Hashtbl.create 20 in
    let add_formal (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);
        ignore (Hashtbl.add var_hash n local)

    in
    List.iter2 add_formal fdecl.sformals (Array.to_list (L.params the_function))
  ;
    (* Return the value for a variable or formal argument.
       Check local names  *)
  let lookup n = Hashtbl.find var_hash n
  in

    (* Construct code for an expression; return its value *)
    let rec expr builder ((_, e) : sexpr) = match e with
      SLiteral i  -> L.const_int i32_t i
      | SFliteral l -> L.const_float_of_string float_t l
      | SNoexpr -> L.const_int i32_t 0
      | SMatrixLit l ->
        let find_inner_type l = match l with
            hd::tl -> let (t,e) = hd in t
          | _ -> A.Int
        in
```

```

      let find_type mat = match mat with
          hd::tl -> find_inner_type hd
        | _ -> A.Int
      in

      let my_type = find_type l in

      let make_matrix = match my_type with
          A.Int ->
          (* extract rows and column info here *)
          let count a = List.fold_left (fun x _ -> x + 1) 0 a in
          let rows = count l in
          let cols = count (List.hd l) in
          let rec valid_dims m = match m with
              hd::tl -> if count hd == count (List.hd l)
                            then valid_dims tl
                          else false
            | _ -> true
          in
          if not (valid_dims l) then
            raise(Failure "all rows of matrices must have the same number of elemens")
          else

          (* allocate space 2 + rows * cols*)
          let matrix = L.build_alloca (L.array_type i32_t (2+rows*cols)) "matrix" builder in

          let eval_row row
            = List.fold_left (fun eval_row x -> eval_row @ [expr builder x]) [] row in
          let unfolded = List.fold_left (fun unfld row -> unfld @ (eval_row row)) [] l in
          let unfolded = [L.const_int i32_t rows; L.const_int i32_t cols] @ unfolded in

          let rec store idx lst = match lst with
            hd::tl -> let ptr = L.build_in_bounds_gep matrix [| L.const_int i32_t 0; L.const_int i32_t idx|] "ptr" builder in
                      ignore(L.build_store hd ptr builder);
                      store (idx + 1) tl;
            | _ -> ()
          in
          store 0 unfolded;
          L.build_in_bounds_gep matrix [|L.const_int i32_t 0; L.const_int i32_t 0|] "matrix" builder
      | A.Float ->
        let count a = List.fold_left (fun x _ -> x + 1) 0 a in
        let rows = float_of_int (count l) in
        let cols = float_of_int (count (List.hd l)) in
        let rec valid_dims m = match m with
            hd::tl -> if count hd == count (List.hd l)
                          then valid_dims tl
                        else false
          | _ -> true
        in
        if not (valid_dims l) then
          raise(Failure "all rows of matrices must have the same number of elemens")
        else

```

```
        (* allocate space 2 + rows * cols*)
        let matrix = L.build_alloca (L.array_type float_t (2+(int_of_float rows)*(int_of_float cols))) "matrix" builder in

        let eval_row row
          = List.fold_left (fun eval_row x -> eval_row @ [expr builder x]) [] row in
        let unfolded = List.fold_left (fun unfld row -> unfld @ (eval_row row)) [] l in
        let unfolded = [L.const_float float_t rows; L.const_float float_t cols] @ unfolded in
        let rec store idx lst = match lst with
          hd::tl -> let ptr = L.build_in_bounds_gep matrix [| L.const_int i64_t 0; L.const_int i64_t idx|] "ptr" builder in
                    ignore(L.build_store hd ptr builder);
                    store (idx + 1) tl;
          | _ -> ()
        in
        store 0 unfolded;
        L.build_in_bounds_gep matrix [| L.const_int i64_t 0; L.const_int i64_t 0|] "matrix" builder
      | _ -> raise (Failure "invalid matrix type")
      in make_matrix
    | SId s       ->
      L.build_load (lookup s) s builder
    | SAssign (s, e) ->
      let e' = expr builder e
      and s' = lookup s in
      let e_type = L.string_of_lltype (L.type_of e')
      and s_type = L.string_of_lltype (L.type_of s') in
      let e_fixed = match (s_type, e_type) with
        "double", "i32" -> L.build_sitofp e' float_t "e_float" builder
        | _ -> e'
    in
      ignore(L.build_store e_fixed s' builder); e'
    | SAccess ((ty, _) as m, r, c) ->
      (* get desired pointer location *)
      let matrix = expr builder m
      and row_idx = expr builder r
      and col_idx = expr builder c in
      let cols = get_matrix_cols matrix builder in
      (* row = row_idx * cols *)
      let row = L.build_mul row_idx cols "row" builder in
      (* row_col = (row_idx * cols) + col_idx *)
      let row_col = L.build_add row col_idx "row_col" builder
      and offset = L.const_int i32_t 2 in
      (* idx = 2 + (row_idx * cols) + col_idx *)
      let idx = L.build_add offset row_col "idx" builder in
      let ptr = L.build_in_bounds_gep matrix [| idx |] "ptr" builder in
      L.build_load ptr "element" builder
    | SBinop ((A.Matrix(A.Int), _) as m1, op, m2) ->
      let m1' = expr builder m1
      and m2' = expr builder m2 in
      let ret = match op with
        A.Add     ->
          L.build_call addm_func [| m1';m2' |] "addm" builder
      | A.Sub     -> L.build_call subm_func [| m1';m2' |] "subm" builder
      | A.Mult    ->
        let (t', _) = m2 in
            let ret_val' = match t' with
              A.Int ->
              let scalar = L.build_sitofp m2' float_t "scalar" builder in
```

```
              L.build_call scalarm_func [| scalar;m1' |] "scalarm" builder
            | A.Float ->
              L.build_call scalarm_func [| m2';m1' |] "scalarm" builder
            | _ ->
              L.build_call multiplication_func [| m1';m2' |] "matm" builder
          in ret_val'
      | A.Equal   -> L.build_call equal_func [| m1';m2' |] "equal" builder
      | A.Neq     -> let eq = L.build_call equal_func [| m1';m2' |] "equal" builder in
                     L.build_xor eq (L.const_int i32_t 1) "and" builder
      | _         -> raise(Failure "internal error: semant should have rejected")
      in ret
    | SBinop ((_ as m1), (_ as op), ((A.Matrix(A.Int),_) as m2)) ->
        let m1' = expr builder m1
        and m2' = expr builder m2 in
        let ret = match op with
        | A.Mult    ->
          let (t, _) = m1 in
          let ret_val = match t with
              A.Int ->
              let scalar = L.build_sitofp m1' float_t "scalar" builder in
              L.build_call scalarm_func [| scalar;m2' |] "scalarm" builder
            | A.Float ->
              L.build_call scalarm_func [| m1';m2' |] "scalarm" builder
            | _ -> raise(Failure "should be caught elsewhere")
          in ret_val
        | _         -> raise(Failure "internal error: semant should have rejected")
        in ret
    | SBinop ((A.Matrix(A.Float), _) as m1, op, m2) ->
      let m1' = expr builder m1
      and m2' = expr builder m2 in
      let ret = match op with
        A.Add     ->
          L.build_call addmf_func [| m1';m2' |] "addmf" builder
      | A.Sub     -> L.build_call submf_func [| m1';m2' |] "submf" builder
      | A.Mult    ->
        let (t', _) = m2 in
            let ret_val' = match t' with
              A.Int ->
              let scalar = L.build_sitofp m2' float_t "scalar" builder in
              L.build_call scalarmf_func [| scalar;m1' |] "scalarmf" builder
            | A.Float ->
              L.build_call scalarmf_func [| m2';m1' |] "scalarmf" builder
            | _ -> L.build_call multiplicationf_func [| m1';m2' |] "matmf" builder
          in ret_val'
      | A.Equal   -> L.build_call equalf_func [| m1';m2' |] "equalf" builder
      | A.Neq     -> let eq = L.build_call equalf_func [| m1';m2' |] "equalf" builder in
                     L.build_xor eq (L.const_int i32_t 1) "and" builder
      | _         -> raise(Failure "internal error: semant should have rejected")
      in ret
    | SBinop ((_ as m1), (_ as op), ((A.Matrix(A.Float),_) as m2)) ->
        let m1' = expr builder m1
        and m2' = expr builder m2 in
        let ret = match op with
        | A.Mult    ->
          let (t, _) = m1 in
          let ret_val = match t with
```

```
                  A.Int ->
                  let scalar = L.build_sitofp m1' float_t "scalar" builder in
                  L.build_call scalarmf_func [| scalar;m2' |] "scalarmf" builder
                | A.Float ->
                  L.build_call scalarmf_func [| m1';m2' |] "scalarm" builder
                | _ -> raise(Failure "should be caught elsewhere")
              in ret_val
            | _          -> raise(Failure "internal error: semant should have rejected")
            in ret
        | SBinop ((t1, e1), op, (t2, e2)) when t1 == A.Float ->
          let e1' = expr builder (t1, e1)
          and e2' = expr builder (t2, e2) in
          let e2' = if t2 == A.Float then e2' else (L.build_uitofp e2' float_t "float_e2" builder) 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
          | _ ->
              raise (Failure "internal error: semant should have rejected and/or on float")
          ) e1' e2' "tmp" builder
        | SBinop ((t1, e1), op, (t2, e2)) when t2 == A.Float ->
          let e1' = expr builder (t1, e1)
          and e2' = expr builder (t2, e2) in
          let e1' = if t1 == A.Float then e1' else (L.build_sitofp e1' float_t "float_e1" builder) 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
          | _ ->
              raise (Failure "internal error: semant should have rejected and/or on float")
          ) e1' e2' "tmp" builder
        | SBinop (e1, op, e2) ->
          let e1' = expr builder e1
          and e2' = expr builder e2 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.Mod     -> L.build_srem
          | 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 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
        | SCall ("print", [e]) | SCall ("printb", [e]) ->
        L.build_call printf_func [| int_format_str ; (expr builder e) |]
          "printf" builder
        | SCall ("printf", [e]) ->
        L.build_call printf_func [| float_format_str ; (expr builder e) |]
          "printf" builder
        | SCall ("printm", [e]) ->
          L.build_call printm_func [| (expr builder e) |] "printm" builder
        | SCall ("printmf", [e]) ->
          L.build_call printmf_func [| (expr builder e)|] "printmf" builder
        | SCall ("getRows", [e]) ->
          let matrix = expr builder e in
          get_matrix_rows matrix builder
        | SCall ("getColumns", [e]) ->
          let matrix = expr builder e in
          get_matrix_cols matrix 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 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 = function
          SBlock sl -> List.fold_left stmt builder sl
        | SExpr e -> ignore(expr builder e); 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) builder );
            builder
        | SIf (predicate, then_stmt, else_stmt) ->
```

```
          let bool_val = expr builder predicate 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) 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) 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) body)
            (L.build_br pred_bb);
          let pred_builder = L.builder_at_end context pred_bb in
          let bool_val = expr pred_builder predicate 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
      | SVarDecl (t, id, e) ->
            let local_var = L.build_alloca (ltype_of_typ t) id builder in
            Hashtbl.add var_hash id local_var;
            let e' = expr builder e in
            let e_type = L.string_of_lltype (L.type_of e')
            and s_type = L.string_of_lltype (ltype_of_typ t) in
            let e_fixed = match (s_type, e_type) with
              "double", "i32" -> L.build_sitofp e' float_t "e_float" builder
              | _ -> e'
          in
            ignore(L.build_store e_fixed (lookup id) builder); builder
      | SUpdate (m, r, c, e) ->
          (* get desired pointer location *)
          let matrix = expr builder m
          and row_idx = expr builder r
          and col_idx = expr builder c in
          let cols = get_matrix_cols matrix builder in
          (* row = row_idx * cols *)
          let row = L.build_mul row_idx cols "row" builder in
          (* row_col = (row_idx * cols) + col_idx *)
          let row_col = L.build_add row col_idx "row_col" builder
          and offset = L.const_int i32_t 2 in
          (* idx = 2 + (row_idx * cols) + col_idx *)
          let idx = L.build_add offset row_col "idx" builder in
          let ptr = L.build_in_bounds_gep matrix [| idx |] "ptr" builder in
          (* update value at that location *)
          let e' = expr builder e in
          let m_typ = L.string_of_lltype (L.type_of matrix)
          and e_typ = L.string_of_lltype (L.type_of e') in
          let e_fixed = match (m_typ, e_typ) with
              "double*", "i32" -> L.build_uitofp e' float_t "float_e" builder
            | "i32*", "double" -> L.build_fptosi e' i32_t "int_e" builder
            | _ -> e'
          in

```

```
            ignore(L.build_store e_fixed ptr builder); builder
      in

      (* Build the code for each statement in the function *)

      let builder = stmt builder (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 functions;
  the_module
```

## 8.3 Parser

Listing 159: mods/parser.mly

```
/* Parser File */
/* Emily, Diego, Cindy, Desu */

/* Ocamlyacc parser for MATRIXMANIA */

%{
 open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK COMMA
%token PLUS MINUS TIMES DIVIDE MOD ASSIGN NOT
%token EQ NEQ LT LEQ GT GEQ AND OR
%token RETURN IF ELIF ELSE FOR WHILE INT FLOAT VOID MATRIX
%token <int> INTLIT
%token <string> ID
%token <string> FLIT
%token <string> STRLIT
%token DEF
%token EOF

%start program
%type <Ast.program> program

%nonassoc NOELSE
%nonassoc NOELIF
%nonassoc ELSE
%nonassoc RETURN
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%left LBRACK RBRACK
%right NOT SIZE

%%

program:
```

```
  fdecls EOF { $1 }

fdecls:
   /* nothing */  { []       }
 | fdecls fdecl   { $2 :: $1 }

fdecl:
   DEF typ ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
     { {
        typ = $2;
        fname = $3;
        formals = List.rev $5;
        body = List.rev $8
     } }

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

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

typ:
   MATRIX LT typ GT { Matrix($3) }
  |INT { Int   }
  | FLOAT { Float }
  | VOID  { Void }

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

block_stmt:
  LBRACE stmt_list RBRACE                    { Block(List.rev $2)    }

elifs:
    ELIF LPAREN expr RPAREN block_stmt %prec NOELSE
                                             { If($3, $5, Block([])) }
  | ELIF LPAREN expr RPAREN block_stmt ELSE block_stmt
                                             { If($3, $5, $7)        }
  | ELIF LPAREN expr RPAREN block_stmt elifs
                                             { If($3, $5, $6)        }

stmt:
    typ ID ASSIGN expr SEMI                  { VarDecl($1, $2, $4)    }
  | expr LBRACK expr COMMA expr RBRACK ASSIGN expr SEMI
                                             { Update($1, $3, $5, $8) }
  | expr SEMI                                { Expr $1                }
  | RETURN expr_opt SEMI                     { Return $2              }
  | block_stmt                               { $1                     }
  | IF LPAREN expr RPAREN block_stmt %prec NOELSE
                                             { If($3, $5, Block([]))  }
  | IF LPAREN expr RPAREN block_stmt ELSE block_stmt
                                             { If($3, $5, $7)         }
  | IF LPAREN expr RPAREN block_stmt elifs  { If($3, $5, $6)         }
  | FOR LPAREN SEMI expr SEMI expr_opt RPAREN stmt
                                       { Block([While($4, Block([$8; (Expr $6)]))])
     }
  | FOR LPAREN stmt expr SEMI expr_opt RPAREN stmt
                                       { Block([$3; While($4, Block([$8; (Expr $6)
    ]))]) }
```

```

  | WHILE LPAREN expr RPAREN stmt           { While($3, $5)          }

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

expr:
    INTLIT           { IntLit($1)            }
  | FLIT             { FLit($1)           }
  | matrix_lit       { MatrixLit($1)          }
  | ID               { Id($1)                 }
  | expr LBRACK expr COMMA expr RBRACK
                     { Access($1, $3, $5)     }
  | 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 MOD    expr { Binop($1, Mod,   $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 ASSIGN expr   { Assign($1, $3)         }
  | ID LPAREN args_opt RPAREN
                     { Call($1, $3)           }
  | LPAREN expr RPAREN
                     { $2                     }

matrix_row:
    expr                   { [$1] }
  | expr COMMA matrix_row  { $1 :: $3 }
/* expr or expr followed by colummn and row, stacking expr, can do math w/ expr
    */

matrix_row_list:
    matrix_row                  { [$1] }
  | matrix_row SEMI matrix_row_list  { $1 :: $3 }
/* 1 row or row followed by semi colon then rest of list */

matrix_lit:
  LBRACK matrix_row_list RBRACK { $2 }
/* list of rows */

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

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

## 8.4 SAST

Listing 160: mods/sast.ml

```
(* Semantically-checked Abstract Syntax Tree and functions for printing it *)
(*Emily, Diego, Sophie, Desu*)
open Ast

type sexpr = typ * sx
and sx =
    SLiteral of int
  | SFliteral of string
  | SMatrixLit of (sexpr list) list
  | SId of string
  | SBinop of sexpr * op * sexpr
  | SUnop of uop * sexpr
  | SAssign of string * sexpr
  | SCall of string * sexpr list
  | SAccess of sexpr * sexpr * sexpr
  | SNoexpr

type sstmt =
    SBlock of sstmt list
  | SVarDecl of typ * string * sexpr
  | SUpdate of sexpr * sexpr * sexpr * sexpr
  | SExpr of sexpr
  | SReturn of sexpr
  | SIf of sexpr * sstmt * sstmt
  | SWhile of sexpr * sstmt

type sfunc_decl = {
    styp : typ;
    sfname : string;
    sformals : bind list;
    sbody : sstmt list;
  }

type sdefine = typ * string * sexpr

type simport = string

type sprogram = sfunc_decl list

(* Pretty-printing functions *)



let rec string_of_sexpr (t, e) =
  "(" ^ string_of_typ t ^ " : " ^ match e with
    SLiteral(l) -> string_of_int l
  | SMatrixLit(l) ->
  let string_of_row l =
    String.concat "" (List.map string_of_sexpr l)
  in
  String.concat "" (List.map string_of_row l)
  | SFliteral(l) -> l
  | SId(s) -> s
  | SBinop(e1, o, e2) ->
      string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2
  | SUnop(o, e) -> string_of_uop o ^ string_of_sexpr e
  | SAssign(v, e) -> v ^ " = " ^ string_of_sexpr e
  | SCall(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
  | SNoexpr -> "(" ^ ")"
  | SAccess(e1, e2, e3) ->
```

```
    string_of_sexpr e1 ^ " " ^ string_of_sexpr e2 ^ " " ^ string_of_sexpr e3

let rec string_of_sstmt = function
    SBlock(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_sstmt stmts) ^ "}\n"
  | SExpr(expr) -> string_of_sexpr expr ^ ";\n";
  | SReturn(expr) -> "return " ^ string_of_sexpr expr ^ ";\n";
  | SIf(e, s1, s2) -> "if (" ^ string_of_sexpr e ^ ")\n" ^
      string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
  | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^ string_of_sstmt s
  | SVarDecl(t, s, e) -> string_of_typ t ^ " " ^ s ^ "=" ^ string_of_sexpr e ^ "\n"
  | SUpdate(m, r, c, e) -> string_of_sexpr e ^ "[" ^ string_of_sexpr e ^ "," ^
                           string_of_sexpr e ^ "] =" ^ string_of_sexpr e ^ "\n"

let string_of_sfdecl fdecl =
  string_of_typ fdecl.styp ^ " " ^
  fdecl.sfname ^ "(" ^ String.concat ", " (List.map snd fdecl.sformals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_sstmt fdecl.sbody) ^
  "}\n"

let string_of_sprogram (funcs) =
  String.concat "\n" (List.map string_of_sfdecl funcs)
```

## 8.5 Scanner

Listing 161: mods/scanner.mll

```
(* Ocamllex scanner for MATRIXMANIA *)
(*Diego, Cindy *)

{ open Parser }

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

(* float literal components*)
let withPoint = digits? '.' digits?
let exponent = 'e' ['+' '-']? digits
let float = withPoint exponent? | digits exponent

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"        { comment lexbuf }           (* Comments *)
| '('         { LPAREN }
| ')'         { RPAREN }
| '{'         { LBRACE }
| '}'         { RBRACE }
| '['         { LBRACK }
| ']'         { RBRACK }
| ';'         { SEMI }
| ','         { COMMA }
| '+'         { PLUS }
| '-'         { MINUS }
| '*'         { TIMES }
| '/'         { DIVIDE }
| '%'         { MOD }
| '='         { ASSIGN }
| "=="        { EQ }
| "!="        { NEQ }
```

```
| '<'         { LT }
| "<="        { LEQ }
| ">"         { GT }
| ">="        { GEQ }
| "&&"        { AND }
| "||"        { OR }
| "!"         { NOT }
| "if"        { IF }
| "else"      { ELSE }
| "elif"      { ELIF }
| "for"       { FOR }
| "while"     { WHILE }
| "return"    { RETURN }
| "int"       { INT }
| "float"     { FLOAT }
| "matrix"    { MATRIX }
| "void"      { VOID }
| "def"       { DEF }
| digits as lxm { INTLIT(int_of_string lxm) }
| float as lxm { FLIT(lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*     as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

## 8.6 Semant

Listing 162: mods/scanner.mll

```
(* Ocamllex scanner for MATRIXMANIA *)
(*Diego, Cindy *)

{ open Parser }

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

(* float literal components*)
let withPoint = digits? '.' digits?
let exponent = 'e' ['+' '-']? digits
let float = withPoint exponent? | digits exponent

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"        { comment lexbuf }           (* Comments *)
| '('         { LPAREN }
| ')'         { RPAREN }
| '{'         { LBRACE }
| '}'         { RBRACE }
| '['         { LBRACK }
| ']'         { RBRACK }
| ';'         { SEMI }
| ','         { COMMA }
| '+'         { PLUS }
| '-'         { MINUS }
| '*'         { TIMES }
| '/'         { DIVIDE }
| '%'         { MOD }
```

```
| '='         { ASSIGN }
| "=="        { EQ }
| "!="        { NEQ }
| '<'         { LT }
| "<="        { LEQ }
| ">"         { GT }
| ">="        { GEQ }
| "&&"        { AND }
| "||"        { OR }
| "!"         { NOT }
| "if"        { IF }
| "else"      { ELSE }
| "elif"      { ELIF }
| "for"       { FOR }
| "while"     { WHILE }
| "return"    { RETURN }
| "int"       { INT }
| "float"     { FLOAT }
| "matrix"    { MATRIX }
| "void"      { VOID }
| "def"       { DEF }
| digits as lxm { INTLIT(int_of_string lxm) }
| float as lxm { FLIT(lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*      as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

## 8.7 matrixmania.ml

Listing 163: mods/matrixmania.ml

```
(* Top-level of the MicroC compiler: scan & parse the input,
   check the resulting AST and generate an SAST from it, generate LLVM IR,
   and dump the module *)

type action = Ast | Sast | LLVM_IR | Compile

let () =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist = [
    ("-a", Arg.Unit (set_action Ast), "Print the AST");
    ("-s", Arg.Unit (set_action Sast), "Print the SAST");
    ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
    ("-c", Arg.Unit (set_action Compile),
      "Check and print the generated LLVM IR (default)");
  ] in
  let usage_msg = "usage: ./matrixmania.native [-a|-s|-l|-c] [file.mc]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;

  let lexbuf = Lexing.from_channel !channel in
  let ast = Parser.program Scanner.token lexbuf in
  match !action with
    Ast -> print_string (Ast.string_of_program ast)
  | _ -> let sast = Semant.check ast in
    match !action with
```

```
      Ast      -> ()
    | Sast     -> print_string (Sast.string_of_sprogram sast)
    | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate sast))
    | Compile -> let m = Codegen.translate sast in
  Llvm_analysis.assert_valid_module m;
  print_string (Llvm.string_of_llmodule m)
```

## 8.8 Makefile

Listing 164: mods/Makefile

```
matrixmania.native: matrixmania.ml codegen.ml semant.ml parser.native scanner.
    native
  ocamlbuild -use-ocamlfind -r matrixmania.native -pkgs llvm,llvm.analysis

parser.native: parser.mly ast.ml scanner.mll
  ocamlbuild -r parser.native

scanner.native: scanner.mll
  ocamlbuild -r scanner.native

printm.o:
  gcc -c c_functions/matrix_functions.c

test: matrixmania.native c_functions/matrix_functions.o
  ./matrixmania.native $(filename) > test.ll
  echo -n "output: "
  llc -relocation-model=pic test.ll
  gcc -o myexe test.s c_functions/matrix_functions.o
  ./myexe
  rm test.ll
  rm myexe
  rm test.s
  rm c_functions/*.o

.PHONY : all
all: clean matrixmania.native c_functions/matrix_functions.o

.PHONY : clean
clean:
  ocamlbuild -clean
  rm -f *.ll
  rm -f *.native
  rm -f parser.ml parser.mli parser.output
  rm -rf _build
  rm -f c_functions/*.o *.o
  rm -f *.exe
  rm -f *.s
  rm -f *.output
```