Matlab Matrix Manipulation

Shenghao Jiang(sj2914)
Shikun Wang(sw3309)
Yixiong Ren (yr2344)
Date: Dec.19 2018

1.Introduction

“MMM” is a Matlab style matrix manipulation language that can be served to perform matrix operations efficiently. The input language of MMM syntactically resembles the C programming language and Matlab style matrix operations. Users are able to do matrix operations through both in-built and self-defined functions. Users are able to define structs, a composite data type, to organize mixed data and multiple matrices for image data.

The application of this programming language ranges from image cropping, rotating, denoising, enhancement, edge detection, and color filtering. Users are able to define the struct and functions to process image more efficiently.

2.Tutorial

2.1 Environment Setup

MMM is developed on Ocaml and LLVM. Please have Ocaml and LLVM installed. Besides, LLVM uses openCV library through C++ program.

2.1.1 Install Homebrew

Homebrew is a free and open-source software package management system that simplifies the installation of software on Apple's macOS operating system.

```
$ /usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

2.1.2 Install openCV

OpenCV (Open source computer vision) is a library of programming functions mainly aimed at real-time computer vision. We will use its function to read, write, and convolve images.

```
$brew install opencv
```

2.1.3 Install pkg-config

pkg-config is a helper tool used when compiling applications and libraries. It helps you insert the correct compiler options on the command line rather than hard-coding values.

```
$brew install pkg-config
```

2.1.4 Install glog

Glog is a logging library for C++.

```
$brew install glog
```

2.2 Run the program

MMM is syntactically similar to C. MMM must contain a main() function, which needs to have 0 as the return value.

2.2.1 Hello World program

Here is a very simple *print.mmm* program, which demonstrates the basic features of MMM.

1. Every program must have a main function with return value 0
2. Every function starts with func as keywords
3. printStr() is a MMM built-in function to print a String value

```
func int main(){
    printStr("hello world");
    return 0;
}
```

To compile the code to generate the executable, run

```
$./compile.sh print.mmm
```

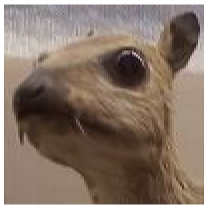
To run the executable, run

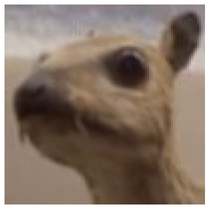
```
$./main
```

The output will be:
hello world

2.2.2 Image blur program

Try to run *ioTest_blut.mmm* to see the blurry effect on the input image *sampleimages/c.png* using openCV library. You will find a blurry image at *sampleimages/c_blur.jpg*.


c.png


c_blur.jpg

To compile the code to generate the executable, run

```
$./compile.sh ioTest_blur.mmm.mmm
```

To run the executable, run

```
$./main
```

3. Language Reference Manual

3.1 Data Types

3.1.1 Basic Data Types

Type Name	Description
int	32-bit signed integer
float	64-bit float point number
bool	bool value for True/False
string	an array of ASCII characters

3.2 Advanced Data Types and Struct

3.2.1 Matrix

Matrix is a data container that can hold only floats. Users can access individual elements in the matrix, or slice the matrix to have a new matrix. Each matrix is regarded as an array and a tuple of integer. The array contains all the elements in the matrix left to right, top to bottom, and the tuple indicates the number of rows and columns of the matrix.

3.2.2 Struct

Structure is a way to group multiple data, which can be in the same type or different type. A struct can be initialized by:

```
struct name {
    type element1;
    type element2;
    …
}
```

Note: there is no semicolon after the last element
Create a struct by:

```
struct img1 = name(arg1,arg2,...)
struct img2 = name(arg1,arg2,...)
```

Once the struct is defined, users can access the data inside struct by:

```
name.element1
name.element2
```

3.3 Lexical Conventions

3.3.1 Identifiers

An identifier of a variable consists of one or more characters where the leading character is a letter followed by a sequence of letter, digits, and possibly underscores.

3.3.2 Keywords

Type Name	Description
if	conditional statement that follows the syntax if-else
else	conditional statement for completing if
for	for-loop follows the syntax for(init;cond;inc)
while	while-loop follows the syntax while(cond){statement}
return	ending current execution and return some values
func	keyword for declaring a function follows syntax func name(arg1,arg2,..)
struct	keyword for declaring a struct that contains data, struct name{type1 ele1; type2 ele2}
matrix	keyword for declaring a matrix contains int, float, matrix name = [...]

int	keyword for declaring a integer, int name = ...
float	keyword for declaring a float, float name = ….
bool	keyword for declaring a bool, bool name = true
string	keyword for declaring a string, string name = “....”
true	boolean type constant
false	boolean type constant

3.3.3 Operations

3.3.3.1 Basic Operators

=	value assignment, a=b
+ - * /	arithmetic operators
-- ++	increment operators
!= == < > <= >=	comparison operators
&& || !	logical operators

3.3.3.2 Matrix Operators

+	addition between matrices
-	subtraction between matrices
*	dot product of two matrices
.*	element-wise matrix multiplication
./	element-wise matrix division
M[a][b]	select the ath row, bth column element. a, b can be expressions
M[:][b], M[a][:]	select all the rows/columns with column/row index, return a matrix. a, b can be expressions
M[i_low:][b], M[a][j_low:]	select row number i_low to end with column number b, return a matrix select column number j_low to end with row number a, return a matrix a, b can be expressions but i_low and j_low have to be integer
M[:i_high][b],	select row number 0 to i_high with column number b, return a matrix

M[a][:j_high]	select column number 0 to j_high with row number a, return a matrix a, b can be expressions but i_high and j_high have to be integer
M[i_low:i_high][b], M[a][j_low:j_high]	select row number i_low to i_high with column number b, return a matrix select column number j_low to j_high with row number a, return a matrix a, b can be expressions but i_low, j_low, i_high, j_high have to be integer
M[i_low:i_high][j_low:j_high]	select row number i_low to i_high with column number j_low to j_high, return a matrix; i_low, j_low, i_high, j_high have to be integer

3.3.4 Comments

inline comment style	// comments	can only comment on one line
multiple comment style	/* comments */	can contain newline character inside

3.4.5 Punctuator

Semicolons at the end of each statement perform no operation but signal the end of a statement.
Statements must be separated by semicolons.

3.4. Syntax

3.4.1 Loops and Statements

3.4.1.1 Conditional Statements

There are three conditional statements: if, else.
The syntax of each of the statement is:

```
if (condition) {statements}
else {statements}
```

There should always be a **if** statement before the **else** statements.
The "{" and "}" besides the statements can be omitted.

3.4.1.2 While loop

The **While** loop syntax is: `while (expr) {stats}`
The "{" and "}" besides statements can be omitted, but each statement must separate by ";"
The loop condition expr can be an integer, a bool, and a comparison expression.

3.4.1.3 For loop

The For loop syntax is: `for(init;cond;inc)`
The ";" that separates each part of the for loop expression cannot be omitted.
The initial variable, conditional statement, and increment method should be a variable, comparison expression, and an expression that increments loop variable.

3.4.2 Expressions

3.4.2.1 Associativity Rules

Tokens (Priority from High to Low)	Associativity
NEG !	Right - Left
* /	Left-Right
.* ./	Left-Right
+ -	Left-Right
> <= < >=	Left-Right
== !=	Left-Right
&&	Left-Right
||	Left-Right
.	Non-Associative
:	Non-Associative
,	Left-Right
Else	Non-Associative
NOELSE	Non-Associative
=	Right - Left
return	Non-Associative
;	Left-Right

3.4.3 Specifications

3.4.3.1 Data Type

There must be a data type specifier in front of each variable name. The data type specifier can be the following:

```
int name = val
float name = val
bool name = val
```

3.4.3.2 Matrix

The matrix specifier identifies a matrix variable.
Example:

```
matrix M[row,col]
```

row and col are integers and it initializes a matrix of zeros with the number of rows and columns

```
matrix M = [1,2;3,4;5,6]
```

“,” is the identifier of elements in different columns and ” ;” is to identify different rows.

3.4.3.3 Functions

Example:

```
func return_type name (type arg1, <struct_name> arg2, ...)
```

When defining a function, the user needs to declare the **func** keyword followed by a return type and a name of the function. In the parenthesis where the inputs are defined, each input argument is separated by “,”. Input arguments include normal data type and the struct that the user has defined.
The function is called by
name (arg1, arg2, ...)
The type of inputs during each call must match the input types when the function is defined.

3.5. Standard Library Functions

3.5.1 Built-in Functions

3.5.1.1 Print functions

Function Name	Description
print(int i)	print a int number
printFloat(float f)	print a float number
printStr(string s)	print a string

3.5.1.2 Matrix calculations

Function Name	Return Type	Description
height(matrix M)	int	return the number of rows in matrix
width(matrix M)	int	return the number of columns in matrix

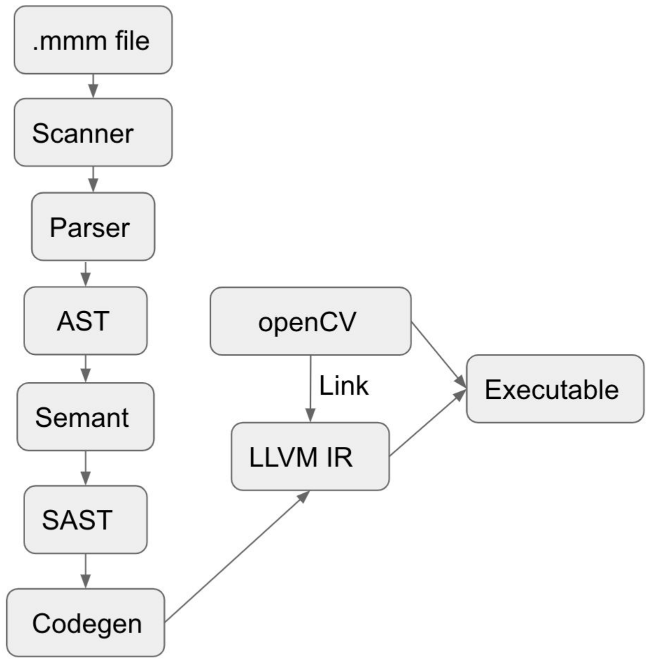
sum(matrix M)	float	return the sum of all the elements in matrix
mean(matrix M)	float	return the average of all the elements in matrix
trans(matrix M)	matrix	transpose a matrix
cov(matrix kernel, matrix M)	matrix	apply a convolutional kernel to a matrix

3.5.3 Image I/O

Function Name	Description
imread(string filepath, struct img, int height, int width)	load a image by giving the file path, image height and width to struct img
imwrite(string filepath, struct img)	save the img struct to the file path
cov_openCV(matrix k, string s, string t, int k_s)	call openCV cov function to convolve kernel k, sized k_s on image in the file path s and save to file path t

4. Translator Architecture

4.1 Architecture Diagram



4.2 Program Structure

mmm.ml, the top level
This is the top level of MMM compiler which invokes ast.ml, sast.ml, scanner.mll, parser.mly, semant.ml, codegen.ml modules to generate the LLVM IR.

parser.mly
Read tokens from the scanner.mll. If no errors, it will generate the AST.

print.ml / sprint.ml
This is the pretty print function used for testing the parser, AST, and SAST.

ast.ml
The abstract syntax tree of the MMM programs.

sast.ml
The semantic checked abstract syntax tree of the MMM programs.

scanner.mll
Scanner will read in the MMM program code and tokenized input code using lexical analysis

semant.ml
The file does semantic checking for before generating LLVM IR. If all the checks passed, it will generate the SAST which is semantically correct.

testall.sh
The file runs all the test cases in tests folder automatically.

codegen.ml
After the semant.ml has checked all the semantic errors, codegen does the work to convert SAST into out files which are LLVM bytecode.

compile.sh
To run the MMM program.

io.cpp
Linked LLVM bytecode with OpenCV to run imread, imwrite, and cov, also handles complex matrix operation.

4.3 Language Evolution

We did many revision on the matrix part. In the first place, we stored Matrix as const_float array into memory, but we decided to move to use struct in LLVM to store Matrix after we considered the tradeoff between language flexibility and compiling speed. Finally, the Matrix is stored as a struct of float array and two integer numbers. We also decided to use openCV as an external library because doing image I/O is difficult and kernel processing is slow with our matrix calculations. After we found the issue of reading image performance, we changed the way to initialize a matrix and optimized our code in image reading and matrix operation, which made image processing much more faster. The use of struct also helps in simplifying the code in our language.

5. Project Plans

5.1 Project Timeline

September 19th	Submitted Proposal	Proposal of building matrix manipulation language
October 15th	Submitted LRM, Parser, and AST	Build the initial parser and ast
November 14th	Submitted Semant, SAST, Hello World Program	Iteration of parser and ast design and add many semantic checkings. The hello World program only prints out int and many undefined.
December 10th	Finished Matrix and Struct Codegen	Finish all parts of matrix and struct, Start working on Image I/O and the demo program, Add a lot of testing on different operations.
December 17th	Finished Image IO	Make some changes of convolutions and finish the first version of demo.

5.2 Style Guide

1. Indentation of each block of OCaml code, especially for let in. One tab distance for thing inside the let in block.
2. Character in each line should be less than 100.
3. Add new-line and comments between each block of code.
4. Follow C-program style in MMM test files.

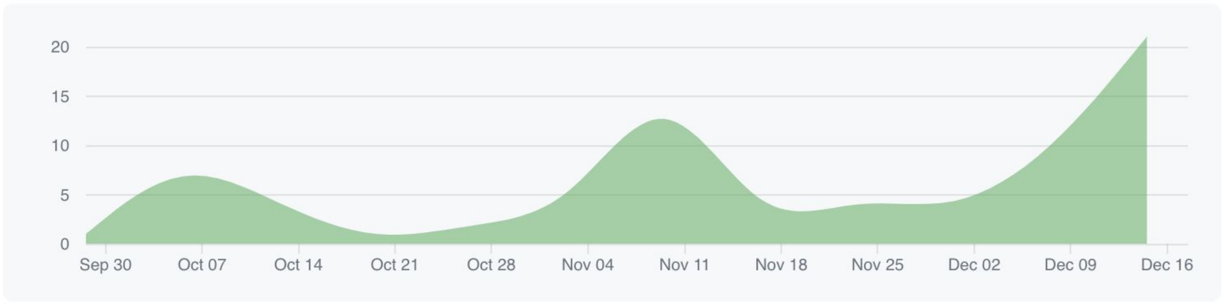
5.3 Team Role

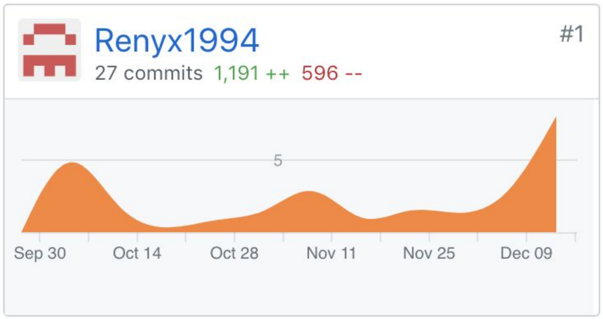
Shenghao Jiang:
Role: Manager and Tester
Work done: test files, demo, parser, make file, io.cpp

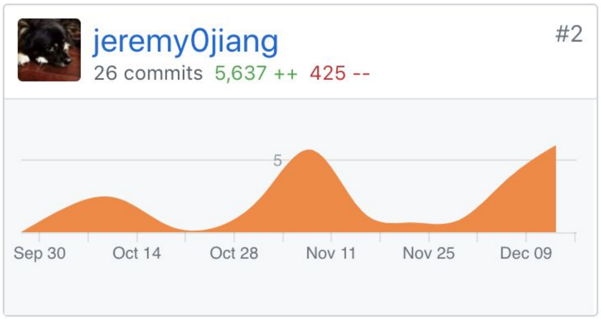
Shikun Wang:
Role: System Architect and Tester
Work done: test files, codegen, scanner, io.cpp

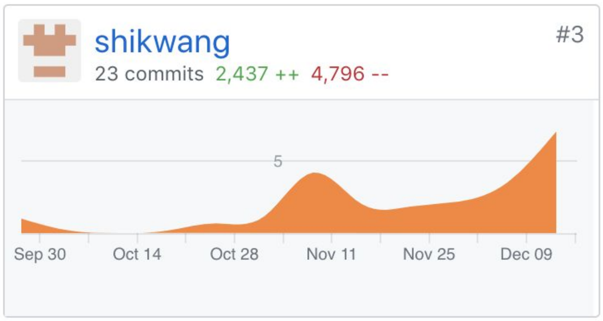
Yixiong Ren:
Role: Language Guru and System Architect
Work done: ast, sast, codegen, semant, scanner

5.4 Project Log









6. Tests

6.1 Source To Target

6.1.1 Function call

Source program:

```
func int main(){
    int a = 1;
    f(a);
    return 0;
}
func void f(int x) {
    print(x);
}
```

Target program:

```
; ModuleID = 'MMM'
source_filename = "MMM"

@fmt = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@fmt.1 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.2 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
@tmp = private unnamed_addr constant [33 x i8] c"Error: Matrix index out of bound\00"
@tmp.3 = private unnamed_addr constant [33 x i8] c"Error: Matrix index out of
bound\00"
@fmt.4 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@fmt.5 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.6 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
@fmt.7 = private unnamed_addr constant [4 x i8] c"%s\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 void @abort()

declare double* @load_cpp(i8*)

declare void @save_cpp(i8*, double*, double*, double*, i32, i32)

declare void @filter_cpp(double*, i8*, i8*, i32)

define void @index_check(i32 %i, i32 %r) {
entry:
  %i1 = alloca i32
  store i32 %i, i32* %i1
  %r2 = alloca i32
  store i32 %r, i32* %r2
  %i3 = load i32, i32* %i1
  %i4 = load i32, i32* %i1
  %tmp = icmp slt i32 %i4, 0
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
  %i5 = load i32, i32* %i1
  %r6 = load i32, i32* %r2
  %i7 = load i32, i32* %i1
  %r8 = load i32, i32* %r2
  %tmp9 = icmp sge i32 %i7, %r8
  br i1 %tmp9, label %then11, label %else13

then:                                             ; preds = %entry
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x
i8]* @fmt, i32 0, i32 0), i8* getelementptr inbounds ([33 x i8], [33 x i8]* @tmp, i32
0, i320))
  call void @abort()
  br label %merge

else:                                             ; preds = %entry
  br label %merge

merge10:                                          ; preds = %else13, %then11
  ret void

then11:                                           ; preds = %merge
  %printf12 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x
i8]* @fmt, i32 0, i32 0), i8* getelementptr inbounds ([33 x i8], [33 x i8]* @tmp.3,
i32 0,i32 0))
  call void @abort()
  br label %merge10

else13:                                           ; preds = %merge
  br label %merge10
}

define i32 @main() {
entry:
  %a = alloca i32
  store i32 1, i32* %a
  %a1 = load i32, i32* %a
  call void @f(i32 %a1)
  ret i32 0
}

define void @f(i32 %x) {
entry:
  %x1 = alloca i32
  store i32 %x, i32* %x1
  %x2 = load i32, i32* %x1
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x
i8]* @fmt.8, i32 0, i32 0), i32 %x2)
  ret void
}
```

6.1.2 Adding two matrix

Source program:

```
func int main()
{
    matrix m1 = [1.1,2.2;3.3,4.4];
    matrix m2 = [1.1,2.2;3.3,4.4];
    matrix res[2,2];
    res = m1+m2;
    float a = res[1][1];
    printFloat(a);
    return 0;
}
```

Target program:

```
; ModuleID = 'MMM'
source_filename = "MMM"

%matrix_t = type { double*, i32, i32 }

@fmt = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@fmt.1 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.2 = private unnamed_addr constant [4 x i8] c"%g\0A\00"
@tmp = private unnamed_addr constant [33 x i8] c"Error: Matrix index out of bound\00"
@tmp.3 = private unnamed_addr constant [33 x i8] c"Error: Matrix index out of
bound\00"
@fmt.4 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@fmt.5 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt.6 = private unnamed_addr constant [4 x i8] c"%g\0A\00"

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

declare void @abort()

declare void @load_cpp(i8*, double*, double*, double*)

declare void @save_cpp(i8*, double*, double*, double*, i32, i32)

declare void @filter_cpp(double*, i8*, i8*, i32)
declare void @iter1mat_cpp(double*, double*, double, i32)

declare void @trans_cpp(double*, double*, i32, i32)

declare void @iter2mat_cpp(double*, double*, double*, i32, i32)

declare void @matmul_cpp(double*, double*, double*, i32, i32, i32, i32)

define void @index_check(i32 %i, i32 %r) {
entry:
  %i1 = alloca i32
  store i32 %i, i32* %i1
  %r2 = alloca i32
  store i32 %r, i32* %r2
  %i3 = load i32, i32* %i1
  %i4 = load i32, i32* %i1
  %tmp = icmp slt i32 %i4, 0
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
  %i5 = load i32, i32* %i1
  %r6 = load i32, i32* %r2
  %i7 = load i32, i32* %i1
  %r8 = load i32, i32* %r2
  %tmp9 = icmp sge i32 %i7, %r8
  br i1 %tmp9, label %then11, label %else13

then:                                             ; preds = %entry
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x
i8]* @fmt, i32 0, i32 0), i8* getelementptr inbounds ([33 x i8], [33 x i8]* @tmp, i32
0, i320))
  call void @abort()
  br label %merge

else:                                             ; preds = %entry
  br label %merge

merge10:                                          ; preds = %else13, %then11
  ret void

then11:                                           ; preds = %merge
  %printf12 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x
i8]* @fmt, i32 0, i32 0), i8* getelementptr inbounds ([33 x i8], [33 x i8]* @tmp.3,
i32 0,i32 0))
  call void @abort()
  br label %merge10

else13:                                           ; preds = %merge
  br label %merge10
}
```

```
define i32 @main() {
entry:
  %a = alloca double
  %m1 = alloca %matrix_t*
  %m2 = alloca %matrix_t*
  %res = alloca %matrix_t*
  %malloccall = tail call i8* @malloc(i32 mul (i32 ptrtoint (double* getelementptr
(double, double* null, i32 1) to i32), i32 4))
  %system_mat = bitcast i8* %malloccall to double*
  %element_ptr = getelementptr double, double* %system_mat, i32 0
  store double 1.100000e+00, double* %element_ptr
  %element_ptr1 = getelementptr double, double* %system_mat, i32 1
  store double 2.200000e+00, double* %element_ptr1
  %element_ptr2 = getelementptr double, double* %system_mat, i32 2
  store double 3.300000e+00, double* %element_ptr2
  %element_ptr3 = getelementptr double, double* %system_mat, i32 3
  store double 4.400000e+00, double* %element_ptr3
  %malloccall4 = tail call i8* @malloc(i32 ptrtoint (%matrix_t* getelementptr
(%matrix_t, %matrix_t* null, i32 1) to i32))
  %m = bitcast i8* %malloccall4 to %matrix_t*
  %m_mat = getelementptr inbounds %matrix_t, %matrix_t* %m, i32 0, i32 0
  store double* %system_mat, double** %m_mat
  %m_r = getelementptr inbounds %matrix_t, %matrix_t* %m, i32 0, i32 1
  store i32 2, i32* %m_r
  %m_c = getelementptr inbounds %matrix_t, %matrix_t* %m, i32 0, i32 2
  store i32 2, i32* %m_c
  store %matrix_t* %m, %matrix_t** %m1
  %malloccall5 = tail call i8* @malloc(i32 mul (i32 ptrtoint (double* getelementptr
(double, double* null, i32 1) to i32), i32 4))
  %system_mat6 = bitcast i8* %malloccall5 to double*
  %element_ptr7 = getelementptr double, double* %system_mat6, i32 0
  store double 1.100000e+00, double* %element_ptr7
  %element_ptr8 = getelementptr double, double* %system_mat6, i32 1
  store double 2.200000e+00, double* %element_ptr8
  %element_ptr9 = getelementptr double, double* %system_mat6, i32 2
  store double 3.300000e+00, double* %element_ptr9
  %element_ptr10 = getelementptr double, double* %system_mat6, i32 3
  store double 4.400000e+00, double* %element_ptr10
  %malloccall11 = tail call i8* @malloc(i32 ptrtoint (%matrix_t* getelementptr
(%matrix_t, %matrix_t* null, i32 1) to i32))
  %m12 = bitcast i8* %malloccall11 to %matrix_t*
  %m_mat13 = getelementptr inbounds %matrix_t, %matrix_t* %m12, i32 0, i32 0
  store double* %system_mat6, double** %m_mat13
  %m_r14 = getelementptr inbounds %matrix_t, %matrix_t* %m12, i32 0, i32 1
  store i32 2, i32* %m_r14
  %m_c15 = getelementptr inbounds %matrix_t, %matrix_t* %m12, i32 0, i32 2
  store i32 2, i32* %m_c15
  store %matrix_t* %m12, %matrix_t** %m2
  %malloccall16 = tail call i8* @malloc(i32 mul (i32 ptrtoint (double* getelementptr
(double, double* null, i32 1) to i32), i32 4))
  %system_mat17 = bitcast i8* %malloccall16 to double*
  %malloccall18 = tail call i8* @malloc(i32 ptrtoint (%matrix_t* getelementptr
(%matrix_t, %matrix_t* null, i32 1) to i32))
  %m19 = bitcast i8* %malloccall18 to %matrix_t*
  %m_mat20 = getelementptr inbounds %matrix_t, %matrix_t* %m19, i32 0, i32 0
  store double* %system_mat17, double** %m_mat20
  %m_r21 = getelementptr inbounds %matrix_t, %matrix_t* %m19, i32 0, i32 1
  store i32 2, i32* %m_r21
  %m_c22 = getelementptr inbounds %matrix_t, %matrix_t* %m19, i32 0, i32 2
  store i32 2, i32* %m_c22
  store %matrix_t* %m19, %matrix_t** %res
  %m123 = load %matrix_t*, %matrix_t** %m1
  %m224 = load %matrix_t*, %matrix_t** %m2
  %m125 = load %matrix_t*, %matrix_t** %m1
  %m226 = load %matrix_t*, %matrix_t** %m2
  %m_mat27 = getelementptr inbounds %matrix_t, %matrix_t* %m125, i32 0, i32 0
  %mat1 = load double*, double** %m_mat27
  %m_mat28 = getelementptr inbounds %matrix_t, %matrix_t* %m226, i32 0, i32 0
  %mat2 = load double*, double** %m_mat28
  %malloccall29 = tail call i8* @malloc(i32 mul (i32 ptrtoint (double* getelementptr
(double, double* null, i32 1) to i32), i32 4))
  %system_mat30 = bitcast i8* %malloccall29 to double*
  %malloccall31 = tail call i8* @malloc(i32 ptrtoint (%matrix_t* getelementptr
(%matrix_t, %matrix_t* null, i32 1) to i32))
  %m32 = bitcast i8* %malloccall31 to %matrix_t*
  %m_mat33 = getelementptr inbounds %matrix_t, %matrix_t* %m32, i32 0, i32 0
  store double* %system_mat30, double** %m_mat33
  %m_r34 = getelementptr inbounds %matrix_t, %matrix_t* %m32, i32 0, i32 1
  store i32 2, i32* %m_r34
  %m_c35 = getelementptr inbounds %matrix_t, %matrix_t* %m32, i32 0, i32 2
  store i32 2, i32* %m_c35
  %m_mat36 = getelementptr inbounds %matrix_t, %matrix_t* %m32, i32 0, i32 0
  %mat = load double*, double** %m_mat36
  call void @iter2mat_cpp(double* %mat1, double* %mat2, double* %mat, i32 0, i32 4)
  store %matrix_t* %m32, %matrix_t** %res
  %res37 = load %matrix_t*, %matrix_t** %res
  %m_mat38 = getelementptr inbounds %matrix_t, %matrix_t* %res37, i32 0, i32 0
  %mat39 = load double*, double** %m_mat38
  %m_r40 = getelementptr inbounds %matrix_t, %matrix_t* %res37, i32 0, i32 1
  %r_mat = load i32, i32* %m_r40
  %m_c41 = getelementptr inbounds %matrix_t, %matrix_t* %res37, i32 0, i32 2
  %c_mat = load i32, i32* %m_c41
  %tmp = mul i32 1, %c_mat
  %index = add i32 1, %tmp
  call void @index_check(i32 1, i32 %r_mat)
  call void @index_check(i32 1, i32 %c_mat)
  %element_ptr_ptr = getelementptr double, double* %mat39, i32 %index
  %element_ptr42 = load double, double* %element_ptr_ptr
  store double %element_ptr42, double* %a
  %a43 = load double, double* %a
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x
i8]* @fmt.6, i32 0, i32 0), double %a43)
  ret i32 0
}

declare noalias i8* @malloc(i32)
```

6.2 Test suites

We have print.ml and spinrt.ml as the "pretty print". These two program will print valid MMM code. we use them to test the scanner and parser in the early stage.

In the tests folder, we have unit-tests for each feature in the program. For every single feature in our program, we have at least two test cases, where one test case should pass and the other test case should fail. We also have integration tests which combine multiple unit test cases together.
We tested the following test levels:

1. Basic data type assignment and operations
2. Control flow
3. Function call
4. Matrix assignment and operations
5. Struct assignment and operations
6. Image IO test

6.3 Test automation

The automation we used is the testall.sh provided in MicroC, and we extend it be suitable for our test cases. The tests that should fail will output to .err file and the tests that should pass will output to .out file. We can view the whole test result in testall.log.
To run the tests, execute the following command in the terminal:
$./testall.sh

6.4 Test roles

Yisiong works on print.ml and sprint.ml. Shenghao and Shikun works on testing codegen. We add tests cases once we have implemented a new feature.

7. Lessons learned

Shenghao Jiang:
I think the most important part I learned is the whole process of building a compiler. At the beginning, we need to use scanner and parser to generate AST. Then, we need to use semant.ml to do semantic check to generate SAST. Afterwards, we can use codegen to generate LLVM IR. Finally, we can build the executable using the LLVM IR and external library. Although we spent the whole semester building a compiler, I think it is worth to understand the principal from building every little piece in the compiler.

Shikun Wang:
I think the most difficult parts of this project is to design a way to efficiently load and store the matrix through llvm. I have learnt a lot by reading previous projects in implementing matrix which gives me a deeper understanding of how matrix been executed in the compiler level. We have made many design decisions and tried different implementation methods to process the matrix. Although the iterative design process was quite time-consuming, it taught me to how to design a language from sketch. Coding in OCaml is also challenging, but I love OCaml after spending many hours writing pattern matching which is magically powerful and efficient.

Yixiong Ren:
In this project we built a basic compiler from scratch and it helped me understand compiler structure and process of compiling a programming language. Using Ocaml to program is tricky at the beginning, but I started to enjoy its ability in pattern matching and list operation as the project went on. The largest problem we encountered in this project is to handle the size of matrix in our language. Instead of always defining and using matrix of constant size, we want our language to be more flexible and “smart” when dealing with matix. Therefore, we tried to let compiler figure out and record matrix dimension by itself and we successfully kept the generality of user-defined functions when they take matrix of arbitrary size as an input. I have learned a lot through this process especially the use of pointers. The use of LLVM also helped me get a better understanding of how assembly language works.

8. Appendix

8.1 scanner.mll

```
{ open Parser }

rule token =
 parse [' ' '\t' '\r' '\n']  { token lexbuf }
     | "/*"                  { comment lexbuf}
     | "//"                  { inlinecom lexbuf}
     | '+'                   { PLUS }
     | '-'                   { MINUS }
     | '*'                   { TIMES }
     | '/'                   { DIVIDE }
     | "++"                  { ADDONE }
     | "--"                  { MINUSONE }
     | ".*"                  { ELETIMES }
     | "./"                  { ELEDIVIDE }
     | '('                   { LPARE }
     | ')'                   { RPARE }
     | '{'                   { LBRACE }
     | '}'                   { RBRACE }
     | '['                   { LBRACK }
     | ']'                   { RBRACK }
     | "&&"                  { AND }
     | "||"                  { OR }
     | '!'                   { NOT }
     | ','                   { COMMA }
     | ';'                   { SEMICOL }
     | ':'                   { COL }
     | '.'                   { DOT }
     | '='                   { ASSIGN }
     | "=="                  { EQUAL }
     | "!="                  { NEQUAL }
     | '>'                   { GT }
     | ">="                  { NLT }
     | '<'                   { LT }
     | "<="                  { NGT }
     | "if"                  { IF }
     | "else"                { ELSE }
     | "elif"                { ELIF }
     | "for"                 { FOR }
     | "while"               { WHILE }
     | "break"               { BREAK }
     | "return"              { RETURN }
     | "func"                { FUNCTION }
     | "struct"              { STRUCT }
     | "matrix"              { MATRIX }
     | "int"                 { INT }
     | "float"               { FLOAT }
     | "string"              { STRING }
     | "void"                { VOID }
     | "bool"                { BOOLEAN }
     | "true"                { TRUE }
     | "false"               { FALSE }
     | ['0'-'9']+ as lit     { INT_LITERAL(int_of_string lit) }
     | ['0'-'9']+'.'['0'-'9']+ as flt    { FLOAT_LITERAL(float_of_string flt) }
     | '"' ([^ '"']* as str) '"'         { STRING_LITERAL(str) }
     | ['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 }
and inlinecom = parse
       ['\r' '\n']           { token lexbuf }
     | _                     { inlinecom lexbuf}
```

8.2 parser.mly

```
%{ open Ast %}

%token PLUS MINUS TIMES DIVIDE
%token ADDONE MINUSONE
%token ELETIMES ELEDIVIDE
%token LPARE RPARE LBRACE RBRACE LBRACK RBRACK
%token AND OR NOT
%token COMMA SEMICOL COL DOT
%token ASSIGN EQUAL NEQUAL
%token GT NLT LT NGT
%token IF ELSE ELIF
%token FOR WHILE BREAK RETURN
%token FUNCTION STRUCT MATRIX
%token INT FLOAT BOOLEAN STRING VOID
%token TRUE FALSE

%token <int> INT_LITERAL
%token <float> FLOAT_LITERAL
%token <string> STRING_LITERAL
%token <string> ID
%token EOF

/* Precedence and associativity of each operator */

%left SEMICOL
%nonassoc RETURN
%right ASSIGN
%nonassoc NOELSE
%nonassoc ELSE
%nonassoc ELIF
%left COMMA
%nonassoc COL
%nonassoc DOT
%left OR
%left AND
%left EQUAL NEQUAL
%left GT NLT LT NGT
%left PLUS MINUS
%left ADDONE MINUSONE
```

```
%left ELETIMES ELEDIVIDE
%left TIMES DIVIDE
%right NOT NEG

%start program
%type <Ast.program> program

%%

program:
 decls EOF { List.rev (fst $1), List.rev (snd $1) }

decls:
   /* nothing */ { [], [] }
 | decls fdecl { ($2 :: fst $1), snd $1 }
 | decls stdecl { fst $1, ($2 :: snd $1) }

fdecl:
 FUNCTION typ ID LPARE formals_opt RPARE LBRACE stmt_list RBRACE
   { { ftyp = $2;
       fname = $3;
       formals = $5;
       body = List.rev $8 } }

stdecl:
 STRUCT ID LBRACE struct_list RBRACE { {stname = $2; stvar = List.rev $4} }

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

formal_list:
   typ ID                          { [Primdecl($1,$2)] }
 | LT ID GT ID                     { [Strudecl($2,$4)] }
 | formal_list COMMA typ ID        { Primdecl($3,$4) :: $1 }
 | formal_list COMMA LT ID GT ID { Strudecl($4,$6) :: $1 }

struct_list:
   typ ID                            { [Primdecl($1,$2)] }
 | struct_list SEMICOL typ ID        { Primdecl($3,$4) :: $1 }

typ:
   INT { Int }
 | FLOAT { Float }
 | BOOLEAN { Boolean }
 | MATRIX { Matrix }
 | STRING { String }
 | VOID { Void }

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

stmt:
   expr_opt SEMICOL { Expr($1) }
 | RETURN expr SEMICOL { Return($2) }
 | LBRACE stmt_list RBRACE { Block(List.rev $2) }
 | IF LPARE expr RPARE stmt %prec NOELSE { If($3, $5, Block([])) }
   /* elseif */
 | IF LPARE expr RPARE stmt ELSE stmt { If($3, $5, $7) }
 | FOR LPARE expr SEMICOL expr SEMICOL expr RPARE stmt  { For($3, $5, $7, $9) }
 | WHILE LPARE expr RPARE stmt { While($3, $5) }
 | typ ID SEMICOL { Initial($1, $2, Empty) }
 | typ ID ASSIGN expr SEMICOL { Initial($1, $2, $4) }
 | typ ID LBRACK INT_LITERAL COMMA INT_LITERAL RBRACK SEMICOL{ Defaultmat($2, $4, $6) }
 | STRUCT ID ASSIGN ID LPARE expr RPARE SEMICOL{ let inputs = match $6 with
                                                             Comma(e1) -> e1
                                                           | Empty -> []
                                                           | _ -> [$6] in IniStrucct($2, $4,
inputs)}

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

expr:
   INT_LITERAL { Intlit($1) }
 | STRING_LITERAL { Stringlit($1) }
 | FLOAT_LITERAL { Floatlit($1) }
 | mat_literal { Matrixlit(fst $1, snd $1) }
 | TRUE { Boolit(true) }
 | FALSE { Boolit(false) }
 | ID { Var($1) }
 | ID DOT ID  { Struaccess ($1, $3)}
 | 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 ADDONE   { Binop($1, Add, Intlit(1)) }
 | expr MINUSONE { Binop($1, Sub, Intlit(1)) }
 | expr ELETIMES expr { Binop($1, Elemult, $3) }
 | expr ELEDIVIDE expr { Binop($1, Elediv, $3) }
 | expr EQUAL expr { Binop($1, Eq, $3) }
 | expr NEQUAL expr { Binop($1, Neq, $3) }
 | expr LT expr { Binop($1, Less, $3) }
 | expr NGT expr { Binop($1, Leq, $3) }
 | expr GT expr { Binop($1, Greater, $3) }
 | expr NLT expr { Binop($1, Geq, $3) }
 | expr AND expr { Binop($1, And, $3) }
 | expr OR expr { Binop($1, Or, $3) }
```

```
 | expr COMMA expr { match $1, $3 with
                      Comma(e1), Comma(e2) -> Comma(e1@e2)
                    | Comma(e1), e2 -> Comma(e1@[e2])
                    | e1, Comma(e2) -> Comma(e1::e2)
                    | e1, e2 -> Comma([e1;e2])}
 | MINUS expr %prec NEG { Uop(Nega, $2) }
 | NOT expr { Uop(Not, $2) }
 | expr ASSIGN expr { Assign($1, $3) }
 | ID LBRACK expr RBRACK LBRACK expr RBRACK { match $3, $6 with
                                              Range(_,_), c -> Matslicing($1, $3, c)
                                            | r, Range(_,_) -> Matslicing($1, r, $6)
                                            | a, b -> Mataccess($1, a, b)
                                            | _ -> failwith("wrong indexing
expression")}
 | ID LPARE expr_opt RPARE { let inputs = match $3 with
                                        Comma(e1) -> e1
                                      | Empty -> []
                                      | _ -> [$3] in Call($1, inputs)}
 | LPARE expr RPARE { $2 }
 | INT_LITERAL COL { Range(Ind($1), End) }
 | INT_LITERAL COL INT_LITERAL { Range(Ind($1), Ind($3)) }
 | COL INT_LITERAL { Range(Beg, Ind($2)) }
 | COL { Range(Beg, End) }

mat_literal:
   LBRACK RBRACK { [| |], (0, 0) }     /* empty matrix */
 | LBRACK mat_assembly RBRACK { $2 }

mat_assembly:
   ele { [| $1 |], (1, 1) }
 | mat_assembly COMMA ele { Array.append (fst $1) [| $3 |], (fst (snd $1) , snd (snd $1) +1)
}
 | mat_assembly SEMICOL ele { Array.append (fst $1) [| $3 |], (fst (snd $1) +1 , 1) }

ele:
   FLOAT_LITERAL { $1 }
 | MINUS FLOAT_LITERAL %prec NEG { -. $2 }
```

8.3 ast.ml

```
type biop = Add | Sub | Mult | Div | Eq | Neq | Less | Leq | Greater | Geq | And | Or |
Elemult | Elediv

(*elemult is matrix between int*)
type uniop = Not | Nega

type datatyp = Int | Float | Boolean | Matrix | String | Void | Struct | SMatrix of int *
int | SStruct of string
```

```
type bind = Primdecl of datatyp * string
 | Strudecl of string * string

type expr =
   Intlit of int
 | Stringlit of string
 | Floatlit of float
 | Boolit of bool
 | Matrixlit of float array * (int * int)
 | Var of string
 | Struaccess of string * string
 | Binop of expr * biop * expr
 | Comma of expr list
 | Assign of expr * expr
 | Uop of uniop * expr
 | Call of string * expr list
 | Mataccess of string * expr * expr
 | Matslicing of string * expr * expr
 | Empty (*declare variable without assigning value*)
 | Range of index * index
and index = Beg | End | Ind of int

type stmt =
   Block of stmt list
 | Expr of expr
 | Return of expr
 | If of expr * stmt * stmt
 | For of expr * expr * expr * stmt
 | While of expr * stmt
 | Initial of datatyp * string * expr
 | Defaultmat of string * int * int
 | IniStrucct of string * string * expr list

type func_decl = {
 mutable ftyp : datatyp;
 fname : string;
 formals : bind list;
 body : stmt list;
}

type struc_decl = {
 stname : string;
 stvar : bind list;
}

type program =  func_decl list * struc_decl list

(* pretty print *)
let string_of_biop = function
   Add -> "+"
 | Sub -> "-"
 | Mult -> "*"
 | Div -> "/"
 | Eq -> "=="
 | Neq -> "!="
 | Less -> "<"
 | Leq -> "<="
 | Greater -> ">"
 | Geq -> ">="
 | And -> "&&"
 | Or -> "||"
 | Elemult -> ".*"
 | Elediv -> "./"

let string_of_uniop = function
   Nega -> "-"
 | Not -> "!"

let rec string_of_expr = function
   Intlit(l) -> string_of_int l
 | Floatlit(l) -> string_of_float l
 | Matrixlit(_,(a,b)) -> "(" ^ string_of_int a ^ ", " ^ string_of_int b ^ ")"
 | Stringlit(l) -> l
 | Boolit(true) -> "true"
 | Boolit(false) -> "false"
 | Var(s) -> s
 | Struaccess(s, p) -> s^ "." ^p
 | Binop(e1, o, e2) -> string_of_expr e1 ^ " " ^ string_of_biop o ^ " " ^ string_of_expr e2
 | Uop(o, e) -> string_of_uniop o ^ string_of_expr e
 | Assign(v, e) -> string_of_expr v ^ " = " ^ string_of_expr e
 | Call(f, el) -> f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
 | Mataccess(m, b, c) -> m ^ "[" ^string_of_expr b^ "][" ^ string_of_expr c ^ "]"
 | Matslicing(m, b, c) -> m ^ "[" ^string_of_expr b^ "][" ^ string_of_expr c ^ "]"
 | Range(s, e) -> let a = (match s with Beg -> "Beg" | End -> "End" | Ind(e) ->
string_of_int e) and b =
                          (match e with Beg -> "Beg" | End -> "End" | Ind(e) ->
string_of_int e) in
                          a ^ " : " ^ b
 | Empty -> ""
 | _ -> ""

let string_of_datatyp = function
   Int -> "int"
 | Boolean -> "bool"
 | Void -> "void"
 | Float -> "float"
 | Matrix -> "matrix"
 | String -> "string"
 | Struct -> "struct"
 | SMatrix(a,b) -> "SMatrix(" ^ string_of_int a ^ ", " ^ string_of_int b ^ ")"
 | _ -> ""

let rec string_of_stmt = function
   Block(stmts) -> "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^"}\n"
 | Expr(expr) -> string_of_expr expr ^ ";\n";
 | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
 | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^string_of_stmt s
 | If(e, s1, s2) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s1 ^ "else\n" ^
string_of_stmt s2
 | For(e1, e2, e3, s) -> "for (" ^ string_of_expr e1 ^ " ; " ^ string_of_expr e2 ^ "; " ^
string_of_expr e3 ^ ") " ^ string_of_stmt s
 | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^string_of_stmt s
 | Initial(t, v, n) -> (match n with
                        Empty -> string_of_datatyp t ^ " " ^ v ^ ";\n"
                       | _ -> string_of_datatyp t^ " " ^ v ^ " = " ^ string_of_expr n ^
";\n")
 | Defaultmat(m, a, b) -> "matrix " ^ m ^ "[" ^ string_of_int a ^ ", " ^ string_of_int b ^
"];\n"
 | IniStrucct(o, s, el) -> "struct " ^ o ^ " = " ^ s ^ "(" ^ String.concat ", " (List.map
string_of_expr el) ^ ");\n"
 | _ -> ""

let string_of_bind = function
   Primdecl(t, v) -> string_of_datatyp t ^ " " ^ v
 | Strudecl(t, v) -> "<" ^ t ^ "> " ^ v

let string_of_stdecl stdecl = "struct " ^ stdecl.stname ^ " {\n" ^ String.concat ";\n"
(List.map string_of_bind stdecl.stvar) ^ "\n}\n"

let string_of_fdecl fdecl = "func " ^ fdecl.fname ^ "(" ^ String.concat ", " (List.map
string_of_bind fdecl.formals) ^
  ")\n{\n" ^ String.concat "" (List.map string_of_stmt fdecl.body) ^ "}\n"

let string_of_program (funcs, structs) = String.concat "\n" (List.map string_of_stdecl
structs) ^ "\n" ^String.concat "\n" (List.map string_of_fdecl funcs)
```

8.4 semant.ml

```
open Ast
open Sast
(* Semantic checking of a program. Returns void if successful,
  throws an exception if something wrong.


  Check each struct, then check each function *)


let check(functions, structures)=
 (* Raise an exception if the given list has a duplicate *)
 let report_duplicate except list =
   let rec helper = function
   n1 :: n2 :: _ when n1 = n2 -> raise(Failure(except n1))
     | _ :: t -> helper t
     | [] -> ()
   in helper (List.sort compare list)
 in

 (* Raise an exception if a given binding is to a void type *)
 let check_not_void exceptf = function
     Primdecl(Void, n) -> raise (Failure (exceptf n))
   | _ -> ()
 in

 (* Raise an exception of the given rvalue type cannot be assigned to
    the given lvalue type *)
 let check_assign lvaluet rvaluet err =
   match lvaluet, rvaluet with
     Matrix, SMatrix(a,b) -> SMatrix(a,b)
   | SMatrix(a,b), Matrix -> SMatrix(a,b)
   | SMatrix(a1,b1), SMatrix(a2,b2) -> if a1 = a2 && b1 = b2 then SMatrix(a1,b1) else raise
(Failure err)
   | _ -> if lvaluet = rvaluet then lvaluet else raise (Failure err)
 in

 (**** Checking struct ****)
 report_duplicate (fun n -> "duplicate struct " ^ n) (List.map (fun st -> st.stname)
structures);

 (**** Checking Functions ****)
 let report_built_in_duplicate list =
   let rec helper = function
   | [] -> ()
   | "print" :: _ -> raise (Failure ("function print may not be defined"))
   | "printStr" :: _ -> raise (Failure ("function printStr may not be defined"))
   | "printFloat" :: _-> raise (Failure ("function printFloat may not be defined"))
   | "height" :: _ -> raise (Failure ("function height may not be defined"))
   | "width" :: _ -> raise (Failure ("function width may not be defined"))
   | "sum" :: _ -> raise (Failure ("function sum may not be defined"))
```

```
    | "mean" :: _ -> raise (Failure ("function mean may not be defined"))
    | "trans" :: _ -> raise (Failure ("function trans may not be defined"))
    | "eig" :: _ -> raise (Failure ("function eig may not be defined"))
    | "inv" :: _ -> raise (Failure ("function inv may not be defined"))
    | "det" :: _ -> raise (Failure ("function det may not be defined"))
    | "cov" :: _ -> raise (Failure ("function cov may not be defined"))
    | "imread" :: _ -> raise (Failure ("function imread may not be defined"))
    | "imwrite" :: _ -> raise (Failure ("function save may not be defined"))
    | _ :: t -> helper t
    in helper list
  in report_built_in_duplicate (List.map (fun fd -> fd.fname) functions);

  (* Add function declaration for a named function *)
  let built_in_decls =
    (* define add name and {func} to map *)
    let part_built_indecls =
      let add_bind map (name,ty,rty) = StringMap.add name {
        ftyp = rty;
        fname = name;
        formals = [Primdecl(ty,"x")];
        body = []
      } map
        (* actually add func to map *)
      in List.fold_left add_bind StringMap.empty [
        ("print",Int,Void);("printStr",String,Void);("printFloat",Float,Void);
        ("height",Matrix,Int);("width",Matrix,Int);
        ("sum",Matrix,Float);("mean",Matrix,Float);("trans",Matrix,Matrix);
        ("eig",Matrix,Void);("inv",Matrix,Matrix);("det",Matrix,Void)
      ]
    in



  let add_bind2 map (name,v) = StringMap.add name v map
    in List.fold_left add_bind2 part_built_indecls [
      ("imread",{
        ftyp = Void;
        fname = "imread";
        formals = [Primdecl(String,"x");
Strudecl("Image","y");Primdecl(Int,"x1");Primdecl(Int,"x2")];
```

```
       body = [];
     });
     ("imwrite",{
       ftyp = Void;
       fname = "imwrite";
       formals = [Primdecl(String,"x"); Strudecl("Image","y")];
       body = [];
     });
     ("cov_openCV",{
       ftyp = Void;
       fname = "cov_openCV";
       formals =
[Primdecl(Matrix,"y");Primdecl(String,"z");Primdecl(String,"x");Primdecl(Int,"x1")];
       body = [];
     });
     ("cov", {
     ftyp = Matrix;
     fname = "cov";
     formals = [Primdecl(Matrix,"x"); Primdecl(Matrix,"y")];
     body = [];
     })]

 in

 let function_decls = List.fold_left (
   fun m fd -> StringMap.add fd.fname fd m)
 built_in_decls functions
 in

 let find_func s =
   try StringMap.find s function_decls
   with Not_found -> raise (Failure ("unrecognized function " ^ s))
 in

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

 let struct_decls = List.fold_left (
   fun m st -> StringMap.add st.stname st m)
   StringMap.empty structures
 in

 let find_str s =
   try StringMap.find s struct_decls
   with Not_found -> raise (Failure ("unrecognized struct " ^ s))
 in

 let check_str str =
   List.iter (check_not_void (fun n -> "illegal void formal " ^ n ^ " in " ^ str.stname))
str.stvar;
   report_duplicate (fun n -> "duplicate formal " ^ n ^ " in " ^ str.stname)
   (List.map (fun b -> match b with
       Primdecl(_,n) -> n
     | Strudecl(_,n) -> n) str.stvar);
   { sstname = str.stname;
     sstvar = str.stvar;}
 in

 let check_function func =
   List.iter (check_not_void (fun n -> "illegal void formal " ^ n ^
     " in " ^ func.fname)) func.formals;

   report_duplicate (fun n -> "duplicate formal " ^ n ^ " in " ^ func.fname)
     (List.map (fun b -> match b with
     Primdecl(_,n) -> n
   | Strudecl(_,n) -> n) func.formals);


   let range_size k a b =  match a,b with
       Beg, End -> k
     | Beg, Ind(s) -> if s <= k then s
                      else raise ( Failure ("Matrix slicing exceeds matrix dimension"))
     | Ind(s), End -> if s <= k then (k - s + 1)
                      else raise ( Failure ("Matrix slicing exceeds matrix dimension"))
     | Ind(sl), Ind(ss) -> if ss <= k then (ss - sl + 1)
                           else raise ( Failure ("Matrix slicing exceeds matrix dimension"))
     | _ -> raise ( Failure ("Illegeal Index"))
   in
   let find_size m mats
iz = try StringMap.find m matsiz
   with Not_found -> raise (Failure ("undefined matrix" ^ m))
   in

   (* Build local symbol table of variables for this function *)
   let (symbols,matrixsize,strmap) =
     let rec build_map sym siz smp = function
       Expr e ->(sym, siz, smp)
     | If(p, b1, b2) -> let (sym1,siz1,smp1) = build_map sym siz smp b1 in build_map sym1 siz1
smp1 b2
     | For(e1, e2, e3, st) -> build_map sym siz smp st
     | While(p, s) -> build_map sym siz smp s
     | Return e -> (sym, siz, smp)
     | Initial(t, v, e) -> if StringMap.mem v sym then raise (Failure "variable has been
defined")
       else (match t with
               Matrix -> (match e with
                            Matrixlit(_,(r,c)) -> (StringMap.add v Matrix sym, StringMap.add
v (r,c) siz, smp)
                          | Var(m) -> (StringMap.add v Matrix sym, StringMap.add v
(find_size m siz) siz, smp)
                          | Matslicing(m, e1, e2) -> let (r,c) = find_size m siz in
                              (match e1, e2 with
                                Range(a1,b1), Range(a2,b2) -> (StringMap.add v Matrix sym,
StringMap.add v (range_size r a1 b1,range_size c a2 b2) siz, smp)
                              | Range(a1,b1), _ -> (StringMap.add v Matrix sym,
StringMap.add v (range_size r a1 b1,1) siz, smp)
                              | _, Range(a2,b2) -> (StringMap.add v Matrix sym,
StringMap.add v (1,range_size c a2 b2) siz, smp))
                          | _ -> raise (Failure "Matrix can not be initialied this way!"))
              | _ -> (StringMap.add v t sym, siz, smp))
     | Defaultmat(m, r, c) -> if StringMap.mem m sym then raise (Failure "matrix has been
defined")
       else (StringMap.add m Matrix sym, StringMap.add m (r,c) siz, smp)
     | IniStrucct(v, stname, elist) -> if StringMap.mem v sym then raise (Failure "struct has
been defined")
       else let sttyp = SStruct(stname) in (StringMap.add v sttyp sym, siz, StringMap.add v
stname smp)
     (* A block is correct if each statement is correct and nothing
     follows any Return statement.  Nested blocks are flattened. *)
     | Block sl ->
         let rec build_map_list a b c = function
           [Return _ as s] -> build_map a b c s
         | Return _ :: _   -> raise (Failure "nothing may follow a return")
         | Block sl :: ss  -> build_map_list a b c (sl @ ss) (* Flatten blocks *)
         | s :: ss         -> let (a1,b1,c1) = build_map a b c s in build_map_list a1 b1 c1 ss
         | []              -> (a, b, c)
         in build_map_list sym siz smp sl
     in
     let (symb, sttmap) =
       let transfer (m,s) f = match f with
       Primdecl(ty,na) -> (StringMap.add na ty m, s)
     | Strudecl(str,na) -> let sttyp = SStruct(str) in (StringMap.add na sttyp m,
StringMap.add na str s)
       in
       List.fold_left transfer (StringMap.empty, StringMap.empty) (func.formals)
     in
     build_map symb StringMap.empty sttmap (Block func.body)
   in


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

   let type_of_struct vn =
     try StringMap.find vn strmap
     with Not_found -> raise (Failure ("unrecognized struct variable " ^ vn))
   in


   (* Return a semantically-checked expression, i.e., with a type *)
   let rec expr = function
       Intlit  l   -> (Int, SIntlit l)
```

```
      | Floatlit l  -> (Float, SFloatlit l)
      | Stringlit l -> (String, SStringlit l)
      | Boolit l    -> (Boolean, SBoolit l)
      | Empty       -> (Void, SEmpty)
      | Matrixlit(l,(r,c)) -> if Array.length l = r * c then (SMatrix(r,c), SMatrixlit(l,
(r,c)))
                              else raise ( Failure ("illegal Matrix Dimension"))
      | Var s       -> let ty = type_of_identifier s in (match ty with
                                Matrix -> let (r,c) = StringMap.find s matrixsize in
(SMatrix(r,c), SVar s)
                              | _ -> (ty, SVar s))
      | Struaccess (vname, member) ->
          let stname = type_of_struct vname in
          let st = find_str stname in
          let mt =
            let rec find x lst = match lst with
                                  [] -> raise (Failure ("unrecognized struct member " ^ x))
                                | hd :: tl -> (match hd with
                                                  Primdecl(ty, nm) -> if nm = x then ty else
find x tl
                                                | _ -> find x tl)
            in find member st.stvar
          in (mt, SStruaccess(vname, member))
      | Assign(var, e) as ex ->
          let (lt, s) = expr var
          and (rt, e') = expr e in
          let err = "illegal assignment " ^ string_of_datatyp lt ^ " = " ^
            string_of_datatyp rt ^ " in " ^ string_of_expr e
          in (match lt,rt,var,e with
             SStruct(a1),SStruct(b1),Var(a),Var(b) ->
                  if a1 = b1 then (SStruct(a1), SAssign((lt, s), (rt, e')))
                  else raise (Failure ("illegal assignment " ^ a1 ^ " = " ^ b1 ^ " in " ^
string_of_expr e))
            | _ -> (check_assign lt rt err, SAssign((lt, s), (rt, e'))))
      | Uop(op, e) as ex ->
          let (t, e') = expr e in
          let err = "illegal unary operator " ^ string_of_uniop op ^ string_of_datatyp t ^ " in
" ^ string_of_expr ex in
          let ty = match op with
```

```
            Nega when t = Int || t = Float || t = Matrix -> t
          | Nega -> (match t with SMatrix(a,b) -> SMatrix(a,b) | _ -> raise (Failure err))
          | Not  when t = Boolean -> Boolean
          | _ -> raise (Failure err)
          in (ty, SUop(op, (t, e')))
      | Binop(e1, op, e2) as e ->
          let (t1, e1') = expr e1
          and (t2, e2') = expr e2 in
          (* All binary operators require operands of the same type *)
          let same = t1 = t2 in
          let err = "illegal binary operator " ^ string_of_datatyp t1 ^ " " ^ string_of_biop op
^ " " ^ string_of_datatyp t2 ^ " in " ^ string_of_expr e
          in
          (* Determine expression type based on operator and operand types *)
          let ty = match op with
            Add | Sub | Mult | Div when same && t1 = Int   -> Int
          | Add | Sub | Mult | Div when same && t1 = Float  -> Float
          | Add | Sub | Mult | Div when t1 = Float && t2 = Int  -> Float
          | Add | Sub | Mult | Div when t1 = Int && t2 = Float  -> Float
          | Mult when t1 = Int || t1 = Float -> (match t2 with SMatrix(a,b) -> SMatrix(a,b) | _
-> raise (Failure err))
          | Mult | Div when t2 = Int || t2 = Float -> (match t1 with SMatrix(a,b) ->
SMatrix(a,b) | _ -> raise (Failure err))
          | Add | Sub |Elemult | Elediv -> (match t1, t2 with SMatrix(a1,b1), SMatrix(a2,b2) ->
                                            if a1 = a2 && b1 = b2 then SMatrix(a1,b1)
                                            else raise (Failure "illegal binary operator for
matrix of different sizes")
                                            | _ -> raise (Failure err))
          | Mult ->(match t1, t2 with SMatrix(a1,b1),SMatrix(a2,b2) ->
                      if b1 = a2 then SMatrix(a1,b2) else raise (Failure "matrix multiplication
fail!")
                      | _ -> raise (Failure err))
          | Eq | Neq              when same                -> Boolean
          | Eq | Neq -> (match t1, t2 with SMatrix(a1,b1),SMatrix(a2,b2) -> Boolean | _ ->
raise (Failure err))
          | Less | Leq | Greater | Geq when same && (t1 = Int || t1 = Float) -> Boolean
          | Less | Leq | Greater | Geq when t1 = Int && t2 = Float   -> Boolean
          | Less | Leq | Greater | Geq when t1 = Float && t2 = Int   -> Boolean
          | And | Or when same && t1 = Boolean -> Boolean
          | _ -> raise ( Failure err)
          in (ty, SBinop((t1, e1'), op, (t2, e2')))
      | Comma(el) -> (Void, SComma(List.map expr el))
      | Mataccess(m, e1, e2) -> let mty = type_of_identifier m in
                                if mty = Matrix then
                                  let (t1, se1) = expr e1 in let (t2, se2) = expr e2 in
                                  (match t1,t2 with
                                    Int,Int -> (Float, SMataccess(m, (t1, se1), (t2, se2)))
                                  | _ -> raise ( Failure ("Index of "^ m ^ " is not a
integer!")))
                                else raise ( Failure (m ^ "is not a matrix!"))
      | Matslicing(m, e1, e2) -> let mty = type_of_identifier m in
                                 if mty = Matrix then let (r,c) = StringMap.find m matrixsize
in
                                 (match e1, e2 with
                                   Range(a1,b1), Range(a2,b2) -> (SMatrix(range_size r a1
b1,range_size c a2 b2), SMatslicing(m, expr e1, expr e2))
                                 | Range(a1,b1), c1 -> let (t1,_) = expr c1 in if t1 = Int then
                                                      (SMatrix(range_size r a1 b1, 1),
SMatslicing(m, expr e1, expr e2))
                                                      else raise ( Failure ("Index of "^ m ^ "
is not a integer!"))
                                 | r1, Range(a2,b2) -> let (t1,_) = expr r1 in if t1 = Int then
                                                      (SMatrix(1, range_size c a2 b2),
SMatslicing(m, expr e1, expr e2))
                                                      else raise ( Failure ("Index of "^ m ^ "
is not a integer!")))
                                 else raise ( Failure (m ^ "is not a matrix!"))


    | Range(a,b) -> (match a,b with
                        Beg, End -> (Int, SRange(SBeg, SEnd))
                      | Beg, Ind(s) -> if s >= 0 then (Int, SRange(SBeg, SInd(s)))
                                       else raise ( Failure ("Illegeal Range"))
                      | Ind(s), End -> if s >= 0 then (Int, SRange(SInd(s),SEnd))
                                       else raise ( Failure ("Illegeal Range"))
                      | Ind(sl), Ind(ss) -> if sl <= ss && sl >=0 && ss >= 0 then (Int,
 SRange(SInd(sl),SInd(ss)))
                                            else raise ( Failure ("Illegeal Range"))
                      | _ -> raise ( Failure ("Illegeal Index")))
       | Call(fname, args) as call ->
           let fd = find_func fname in
```

```
         let param_length = List.length fd.formals in
         if List.length args != param_length then
           raise (Failure ("expecting " ^ string_of_int param_length ^
                           " arguments in " ^ string_of_expr call))
         else
         let check_call ff e = let (et, e') = expr e in
           let err = "illegal argument found " ^ string_of_datatyp et ^ " in " ^
string_of_expr e
           in match ff with
               Primdecl(styp,_) -> (check_assign styp et err, e')
             | Strudecl(strty,_) -> (match et with
                                       SStruct(stn) -> if stn = strty then (SStruct(stn),
e')
                                       else raise (Failure ("inconsistent type of struct " ^
strty ^ " and " ^ stn ^ " in " ^ string_of_expr e))
                                     | _ -> raise (Failure ("illegeal type of struct ")))
         in
         let args' = List.map2 check_call fd.formals args
         in (fd.ftyp, SCall(fname, args'))
   in

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

   (* Return a semantically-checked statement i.e. containing sexprs *)
   let rec check_stmt = function
     Expr e -> SExpr (expr e)
   | If(p, b1, b2) -> SIf(check_bool_expr p, check_stmt b1, check_stmt b2)
   | For(e1, e2, e3, st) -> SFor(expr e1, check_bool_expr e2, expr e3, check_stmt st)
   | While(p, s) -> SWhile(check_bool_expr p, check_stmt s)
   | Return e -> let (t, e') = expr e in (match t with
         SMatrix(a,b) -> if func.ftyp = Matrix then SReturn (t, e') else raise (Failure (
           "return gives " ^ string_of_datatyp t ^ " expected " ^ string_of_datatyp
func.ftyp ^ " in " ^ string_of_expr e))
        | _ -> if t = func.ftyp then SReturn (t, e')
     else raise (Failure ("return gives " ^ string_of_datatyp t ^ " expected " ^
     string_of_datatyp func.ftyp ^ " in " ^ string_of_expr e)))

   | Initial(t, v, e) -> let (t', e') = expr e in
     let err = "Initial gives " ^ string_of_datatyp t' ^ " expected " ^
        string_of_datatyp t ^ " in " ^ string_of_expr e in
     (match t, t' with
        Matrix, SMatrix(a,b) -> SInitial(t', v, (t', e'))
      | _ ->
     if t' = t || e = Empty then match t with
       SStruct(_) | Void  -> raise ( Failure (string_of_datatyp t ^ " cannot be initialed
this way!"))
```

```
      | _ -> SInitial(t, v, (t', e'))
      else raise ( Failure err))

    | Defaultmat(m, r, c) -> SDefaultmat(m, r, c)

    | IniStrucct(v, stname, elist) ->
      let st = find_str stname in
      let mem_length = List.length st.stvar in
      if List.length elist != mem_length then
      raise (Failure ("expecting " ^ string_of_int mem_length ^
                      " members in " ^ stname ))
      else let check_mem sv e = match sv with Primdecl(styp,_) ->
        let (et, e') = expr e in
        let err = "illegal argument found " ^ string_of_datatyp et ^
        " expected " ^ string_of_datatyp styp ^ " in " ^ string_of_expr e
        in (check_assign styp et err, e')
      in
      let elist' = List.map2 check_mem st.stvar elist in SIniStrucct(v, stname, elist')
    (* A block is correct if each statement is correct and nothing
       follows any Return statement.  Nested blocks are flattened. *)
    | Block sl ->
        let rec check_stmt_list = function
            [Return _ as s] -> [check_stmt s]
          | Return _ :: _   -> raise (Failure "nothing may follow a return")
          | Block sl :: ss  -> check_stmt_list (sl @ ss) (* Flatten blocks *)
          | s :: ss         -> check_stmt s :: check_stmt_list ss
          | []              -> []
        in SBlock(check_stmt_list sl)
    in (* body of check_function *)
      { sftyp = func.ftyp;
        sfname = func.fname;
        sformals = func.formals;
        slocals = List.map (fun (v,ty) -> Primdecl(ty,v)) (StringMap.bindings symbols);
        smatsiz = StringMap.bindings matrixsize;
        strlist = StringMap.bindings strmap;
        sbody = match check_stmt (Block func.body) with
        SBlock(sl) -> sl
        | _ -> raise (Failure ("internal error: block didn't become a block?"))
      }
  in (List.map check_function functions, List.map check_str structures)
```

8.5 sast.ml

```
(* Semantically-checked Abstract Syntax Tree and functions for printing it *)

open Ast
```

```
module StringMap = Map.Make(String)

type sexpr = datatyp * sx
and sx =
   SIntlit of int
  | SStringlit of string
  | SFloatlit of float
  | SBoolit of bool
  | SMatrixlit of float array * (int * int)
  | SVar of string
  | SStruaccess of string * string
  | SBinop of sexpr * biop * sexpr
  | SComma of sexpr list
  | SAssign of sexpr * sexpr
  | SUop of uniop * sexpr
  | SCall of string * sexpr list
  | SMataccess of string * sexpr * sexpr
  | SMatslicing of string * sexpr * sexpr
  | SEmpty (*declare variable without assigning value*)
  | SRange of sindex * sindex
and sindex = SBeg | SEnd | SInd of int

type sstmt =
   SBlock of sstmt list
  | SExpr of sexpr
  | SReturn of sexpr
  | SIf of sexpr * sstmt * sstmt
  | SFor of sexpr * sexpr * sexpr * sstmt
  | SWhile of sexpr * sstmt
  | SInitial of datatyp * string * sexpr
  | SDefaultmat of string * int * int
  | SIniStrucct of string * string * sexpr list

type sfunc_decl = {
   mutable sftyp : datatyp;
   sfname : string;
   sformals : bind list;
   slocals : bind list;
   smatsiz: (string * (int * int)) list;
   strlist: (string * string) list;
   sbody : sstmt list;
 }

type sstruc_decl = {
   sstname : string;
   sstvar : bind list;
 }

type sprogram =  sfunc_decl list * sstruc_decl list
```

```
(* Pretty-printing functions *)
let rec string_of_sexpr = function
   (_, SIntlit(l)) -> string_of_int l
 | (_, SFloatlit(l)) -> string_of_float l
 | (_, SMatrixlit(_,(a,b))) -> "(" ^ string_of_int a ^ ", " ^ string_of_int b ^ ")"
 | (_, SStringlit(l)) -> l
 | (_, SBoolit(true)) -> "true"
 | (_, SBoolit(false)) -> "false"
 | (_, SVar(s)) -> s
 | (_, SStruaccess(s, p)) -> s^ "." ^p
 | (_, SBinop(e1, o, e2)) -> string_of_sexpr e1 ^ " " ^ string_of_biop o ^ " " ^
string_of_sexpr e2
 | (_, SUop(o, e)) -> string_of_uniop o ^ string_of_sexpr e
 | (_, SAssign(v, e)) -> string_of_sexpr v ^ " = " ^ string_of_sexpr e
 | (_, SCall(f, el)) -> f ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
 | (_, SMataccess(m, b, c)) -> m ^ "[" ^string_of_sexpr b^ "][" ^ string_of_sexpr c ^ "]"
 | (_, SMatslicing(m, b, c)) -> m ^ "[" ^string_of_sexpr b^ "][" ^ string_of_sexpr c ^ "]"
 | (_, SRange(s, e)) -> let a = (match s with SBeg -> "SBeg" | SEnd -> "SEnd" | SInd(e) ->
string_of_int e) and b =
                        (match e with SBeg -> "SBeg" | SEnd -> "SEnd" | SInd(e) ->
string_of_int e) in
                        a ^ " : " ^ b
 | (_, SEmpty) -> ""
 | _ -> ""

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, s, SBlock([])) -> "if (" ^ string_of_sexpr e ^ ")\n" ^string_of_sstmt s
 | SIf(e, s1, s2) -> "if (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt s1 ^ "else\n" ^
string_of_sstmt s2
 | SFor(e1, e2, e3, s) -> "for (" ^ string_of_sexpr e1 ^ " ; " ^ string_of_sexpr e2 ^ "; " ^
string_of_sexpr e3 ^ ") " ^ string_of_sstmt s
 | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^string_of_sstmt s
 | SInitial(t, v, n) -> (match n with
                           (Void, SEmpty) -> string_of_datatyp t ^ " " ^ v ^ ";\n"
                          | _ -> string_of_datatyp t^ " " ^ v ^ " = " ^ string_of_sexpr n ^
";\n")
 | SDefaultmat(m, a, b) -> "matrix " ^ m ^ "[" ^ string_of_int a ^ ", " ^ string_of_int b ^
"];\n"
 | SIniStrucct(o, s, el) -> "struct " ^ o ^ " = " ^ s ^ "(" ^ String.concat ", " (List.map
string_of_sexpr el) ^ ");\n"
 | _ -> ""

let string_of_sstdecl stdecl = "struct " ^ stdecl.sstname ^ " {\n" ^ String.concat ";\n"
(List.map string_of_bind stdecl.sstvar) ^ "\n}\n"

let string_of_sfdecl fdecl = "func " ^ string_of_datatyp fdecl.sftyp ^ " " ^fdecl.sfname ^
"(" ^
   String.concat ", " (List.map string_of_bind fdecl.sformals) ^ ")\n{\n" ^ String.concat ""
(List.map string_of_sstmt fdecl.sbody) ^ "}\n"

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

8.6 codegen.ml

```
module L = Llvm
module A = Ast
open Sast
open Ast

module StringMap = Map.Make(String)

(* translate : Sast.program -> Llvm.module *)
let translate (functions, structs) =
 let idx_check = {
         sftyp = Void;
         sfname = "index_check";
         sformals = [Primdecl(Int,"i");Primdecl(Int,"r")];
         slocals = [];
         smatsiz = [];
         strlist =[];
         sbody = [SIf((Boolean, SBinop ((Int, SVar "i"), Less, (Int, SIntlit 0))),
SBlock([SExpr (Void, SCall("abort",[]))]), SBlock([]));
         SIf((Boolean, SBinop ((Int, SVar "i"), Geq, (Int, SVar "r"))), SBlock([SExpr (Void,
SCall("abort",[]))]), SBlock([]))]
 }
 in
 let functions = idx_check::functions in

 let context    = L.global_context () in

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

 (* Get types from the context *)
 let i32_t      = L.i32_type    context
 and i8_t       = L.i8_type     context
 and i1_t       = L.i1_type     context
 and float_t    = L.double_type context
 and void_t     = L.void_type   context
 and array_t    = L.array_type
 and pointer_t  = L.pointer_type
 in
 let matrix_t = L.named_struct_type context "matrix_t" in
   L.struct_set_body matrix_t [|L.pointer_type float_t; i32_t; i32_t|] false;

 (*let matrixptr_t = L.pointer_type matrix_t in*)

 (* Return the LLVM type for a MicroC type *)
 (* To do : matrix and struct *)
 let ltype_of_typ = function
       A.Int   -> i32_t
     | A.Boolean  -> i1_t
     | A.Float -> float_t
     | A.Void  -> void_t
     | A.String  -> pointer_t i8_t
     (*
     | A.Matrix -> array_t float_t 4 (*the int must be equal to total size of the matrix*)
     *)
     | A.Matrix -> L.pointer_type matrix_t
 in

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


 (* use to interrupt the function flow and throw run-time exception *)
 let abort_func = L.declare_function "abort" (L.function_type void_t [||]) the_module in


 let load_cpp_t : L.lltype = L.function_type void_t [| L.pointer_type i8_t;L.pointer_type
float_t;L.pointer_type float_t;L.pointer_type float_t |] in
 let load_cpp_func : L.llvalue = L.declare_function "load_cpp" load_cpp_t the_module in

 let save_cpp_t : L.lltype = L.function_type void_t [| L.pointer_type i8_t; L.pointer_type
float_t;
                              L.pointer_type float_t; L.pointer_type float_t; i32_t; i32_t;
|] in
 let save_cpp_func : L.llvalue = L.declare_function "save_cpp" save_cpp_t the_module in

 let filter_cpp_t : L.lltype = L.function_type void_t [| L.pointer_type float_t;
L.pointer_type i8_t; L.pointer_type i8_t; i32_t|] in
 let filter_cpp_func : L.llvalue = L.declare_function "filter_cpp" filter_cpp_t the_module
in

 let iter1mat_cpp_t : L.lltype = L.function_type void_t [| L.pointer_type float_t;
L.pointer_type float_t; float_t; i32_t|] in
 let iter1mat_cpp_func : L.llvalue = L.declare_function "iter1mat_cpp" iter1mat_cpp_t
the_module in

 let trans_cpp_t : L.lltype = L.function_type void_t [| L.pointer_type float_t;
L.pointer_type float_t; i32_t; i32_t|] in
 let trans_cpp_func : L.llvalue = L.declare_function "trans_cpp" trans_cpp_t the_module in

 let iter2mat_cpp_t : L.lltype = L.function_type void_t [| L.pointer_type float_t;
L.pointer_type float_t; L.pointer_type float_t; i32_t; i32_t|] in
 let iter2mat_cpp_func : L.llvalue = L.declare_function "iter2mat_cpp" iter2mat_cpp_t
the_module in

 let matmul_cpp_t : L.lltype = L.function_type void_t [| L.pointer_type float_t;
L.pointer_type float_t; L.pointer_type float_t; i32_t; i32_t; i32_t; i32_t|] in
 let matmul_cpp_func : L.llvalue = L.declare_function "matmul_cpp" matmul_cpp_t the_module
in


 let struct_decls : (L.lltype * sstruc_decl) StringMap.t =
   let struct_decl m sdecl =
     let struc_name = sdecl.sstname
         and mem_types = Array.of_list(List.map (fun (t,_) -> ltype_of_typ t)
         (List.map (fun c -> match c with
           Primdecl(a,b) -> (a,b)
         | _ -> raise (Failure "Struct cannot have struct member!")) sdecl.sstvar))
     in let stype = L.named_struct_type context struc_name in
     ignore(L.struct_set_body stype mem_types false);
     StringMap.add struc_name (stype, sdecl) m in
   List.fold_left struct_decl StringMap.empty structs in

 (* Define each function (arguments and return type) so we can
   call it even before we've created its body *)
   let function_decls : (L.llvalue * sfunc_decl) StringMap.t =
     let function_decl m fdecl =
       let name = fdecl.sfname
       and formal_types =
           let formal_list = List.map (fun c -> match c with
           Primdecl(a,b) -> (a,b)
         | Strudecl(a,b) -> (SStruct(a),b)) fdecl.sformals
           in let typ_trans (t,v) = match t with
               SStruct(stn) -> let (sdef,_) = StringMap.find stn struct_decls in
                               pointer_t sdef
             | _ -> ltype_of_typ t
           in
           Array.of_list (List.map typ_trans formal_list)
       in let ftype = L.function_type (ltype_of_typ fdecl.sftyp) 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 string_format_str = L.build_global_stringptr "%s\n" "fmt" builder
    and 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 local_vars =
      let add_formal m (t, n) p =
        L.set_value_name n p;
         let local = match t with
                    SStruct(stn) -> let (sdef,_) = StringMap.find stn struct_decls in
                         L.build_alloca (pointer_t sdef) (n^"_ptr") builder
                  | _ -> L.build_alloca (ltype_of_typ t) n builder
        in
      ignore (L.build_store p local builder);
       StringMap.add n local m

      (* Allocate space for any locally declared variables and add the
      * resulting registers to our map *)
      and add_local m (t, n) =
          let local_var = match t with
              SStruct(stn) -> let (sdef,_) = StringMap.find stn struct_decls in
                   L.build_alloca (pointer_t sdef) (n^"_ptr") builder
            | _ -> L.build_alloca (ltype_of_typ t) n builder
          in StringMap.add n local_var m
      in
      let formal_list = List.map (fun c -> match c with
                                            Primdecl(a,b) -> (a,b)
                                          | Strudecl(a,b) -> (SStruct(a),b)) fdecl.sformals
      in
      let formals = List.fold_left2 add_formal StringMap.empty formal_list
                    (Array.to_list (L.params the_function))
      in List.fold_left add_local formals (List.filter (fun s -> not (List.mem s
formal_list))
                    (List.map (fun c -> match c with
                      Primdecl(a,b) -> (a,b)
                    | Strudecl(a,b) -> (SStruct(a),b))  fdecl.slocals))
    in

    (* Return the value for a variable or formal argument.
       Check local names first, then global names *)
    let lookup n = StringMap.find n local_vars
    in

    let stru_name =
      List.fold_left (fun tmp c -> match c with
      | (v,stn) -> StringMap.add v stn tmp) StringMap.empty fdecl.strlist
    in
```

```
   let mat_sizes =
     List.fold_left (fun tmp c -> match c with
     | (s,(r,c)) -> StringMap.add s (r,c) tmp) StringMap.empty fdecl.smatsiz
   in

   let lookup_size n = match n with
     | (A.SMatrix (r,c),_) -> (r,c)
     | _ -> (0,0)
   in

   let find_size_inmap n =
     try StringMap.find n mat_sizes with
     Not_found -> raise(Failure("Not found the matrix size"))
   in

   (*the function builds the matrixlit*)
   let build_matrix_lit(f_array,(r,c)) builder =
     let mat = L.build_array_malloc float_t (L.const_int i32_t (r*c)) "system_mat" builder
in
     (*try cast pointer*)
     (for x = 0 to (r*c-1) do
       let element_ptr = L.build_gep mat [|(L.const_int i32_t x)|] "element_ptr" builder in
       ignore(L.build_store (L.const_float float_t f_array.(x)) element_ptr builder)
     done);
     let m = L.build_malloc matrix_t "m" builder in
     let m_mat = L.build_struct_gep m 0 "m_mat" builder in ignore(L.build_store mat m_mat
builder);
     let m_r = L.build_struct_gep m 1 "m_r" builder in ignore(L.build_store (L.const_int
i32_t r) m_r builder);
     let m_c = L.build_struct_gep m 2 "m_c" builder in ignore(L.build_store (L.const_int
i32_t c) m_c builder);
     m (*L.build_pointercast m matrixptr_t "m" builder*)
   in

   let build_default_mat (r,c) builder =
     let mat = L.build_array_malloc float_t (L.const_int i32_t (r*c)) "system_mat" builder
in
     (*try cast pointer*)
     (*for x = 0 to (r*c-1) do
       let element_ptr = L.build_gep mat [|(L.const_int i32_t x)|] "element_ptr" builder in
       ignore(L.build_store (L.const_float float_t 0.0) element_ptr builder)
     done*)
     let m = L.build_malloc matrix_t "m" builder in
     let m_mat = L.build_struct_gep m 0 "m_mat" builder in ignore(L.build_store mat m_mat
builder);
     let m_r = L.build_struct_gep m 1 "m_r" builder in ignore(L.build_store (L.const_int
i32_t r) m_r builder);
     let m_c = L.build_struct_gep m 2 "m_c" builder in ignore(L.build_store (L.const_int
i32_t c) m_c builder);
     m (*L.build_pointercast m matrixptr_t "m" builder*)
   in

   let is_matrix ptr =
     let ltype_string = L.string_of_lltype (L.type_of ptr) in
     match ltype_string with
       "%matrix_t*" -> true
     | _ -> false
   in

   let mat_float_operation m1 r1 c1 num builder =
     let mat1 = L.build_load (L.build_struct_gep m1 0 "m_mat" builder) "mat1" builder in
     let res_mat = build_default_mat (r1,c1) builder in (* Change the r and c size here *)
     let res = L.build_load (L.build_struct_gep res_mat 0 "m_mat" builder) "mat" builder in
     ignore(L.build_call iter1mat_cpp_func [| mat1;res;num;(L.const_int i32_t (r1*c1)) |] ""
builder);
     (*for i = 0 to r1*c1-1 do
       let m1_ele_ptr_ptr = L.build_gep mat1 [|L.const_int i32_t i|] "element_ptr_ptr"
builder in
       let m1_ele_ptr = L.build_load m1_ele_ptr_ptr "element_ptr" builder in
       let res_ptr_ptr = L.build_gep res [|L.const_int i32_t i|] "res_ptr_ptr" builder in
       let tmp_ele = L.build_fmul m1_ele_ptr num "tmp_ele" builder in ignore(L.build_store
tmp_ele res_ptr_ptr builder)
     done*); res_mat
   in

   let mat_mat_operation m1 m2 r1 c1 r2 c2 op if_elewise builder =
     match if_elewise with
     | "yes" ->
       let mat1 = L.build_load (L.build_struct_gep m1 0 "m_mat" builder) "mat1" builder in
       let mat2 = L.build_load (L.build_struct_gep m2 0 "m_mat" builder) "mat2" builder in
       let res_mat = build_default_mat (r1,c1) builder in (* Change the r and c size here *)
       let res = L.build_load (L.build_struct_gep res_mat 0 "m_mat" builder) "mat" builder
in
       ignore(L.build_call iter2mat_cpp_func [| mat1;mat2;res;(L.const_int i32_t
op);(L.const_int i32_t (r1*c1)) |] "" builder);
       res_mat

     | "not" ->
       let mat1 = L.build_load (L.build_struct_gep m1 0 "m_mat" builder) "mat1" builder in
       let mat2 = L.build_load (L.build_struct_gep m2 0 "m_mat" builder) "mat2" builder in
       let res_mat = build_default_mat (r1,c2) builder in
       let res = L.build_load (L.build_struct_gep res_mat 0 "m_mat" builder) "mat" builder
in
       ignore(L.build_call matmul_cpp_func [| mat1;mat2;res;(L.const_int i32_t
r1);(L.const_int i32_t c1);(L.const_int i32_t r2);(L.const_int i32_t c2) |] "" builder);
       res_mat
   in

   (* Construct code for an expression; return its value *)
   let rec expr builder ((typ, e) : sexpr) = match e with
```

```
        SIntlit i  -> L.const_int i32_t i
      | SBoolit b  -> L.const_int i1_t (if b then 1 else 0)
      | SFloatlit l -> L.const_float float_t l
      | SStringlit s -> L.build_global_stringptr s "tmp" builder

      | SMatrixlit (f_array,(r,c)) -> (build_matrix_lit (f_array,(r,c)) builder)
      | SMataccess (s,e1,e2) ->
        let idx =
          let e1' = expr builder e1 and e2' = expr builder e2 in
          let ptr = L.build_load (lookup s) s builder in
          let mat = L.build_load (L.build_struct_gep ptr 0 "m_mat" builder) "mat" builder in
          let r = L.build_load (L.build_struct_gep ptr 1 "m_r" builder) "r_mat" builder in
          let c = L.build_load (L.build_struct_gep ptr 2 "m_c" builder) "c_mat" builder in
          let index = L.build_add e2' (L.build_mul e1' c "tmp" builder) "index" builder in
          let (fdef, _) = StringMap.find "index_check" function_decls in
          ignore(L.build_call fdef [| e1'; r |] "" builder);
          ignore(L.build_call fdef [| e2'; c |] "" builder);
          L.build_gep mat [|index|] "element_ptr_ptr" builder
        in
        L.build_load idx "element_ptr" builder

      (* add matrix slicing here*)
      | SMatslicing(s,e1,e2) ->
        (match (e1,e2) with
          | ((_,SRange(rs1,rt1)),(_,SRange(rs2,rt2))) ->
            (let (r,c) = find_size_inmap s in

            let (rs1,rt1) = (match e1 with (_,SRange(rs1,rt1)) -> (rs1,rt1)) in
            let s1 = match rs1 with SBeg -> 0 | SEnd -> r-1 | SInd(s1) -> s1 in
            let t1 = match rt1 with SBeg -> 0 | SEnd -> r-1 | SInd(t1) -> t1 in

            let (rs2,rt2) = (match e2 with (_,SRange(rs2,rt2)) -> (rs2,rt2)) in
            let s2 = match rs2 with SBeg -> 0 | SEnd -> c-1 | SInd(s2) -> s2 in
            let t2 = match rt2 with SBeg -> 0 | SEnd -> c-1 | SInd(t2) -> t2 in

            let ptr = L.build_load (lookup s) s builder in
            let mat = L.build_load (L.build_struct_gep ptr 0 "m_mat" builder) "mat" builder
in

            let res_mat = build_default_mat ((t1-s1+1),(t2-s2+1)) builder in
            let res = L.build_load (L.build_struct_gep res_mat 0 "m_mat" builder) "mat"
builder in

            let pointer = ref 0 in

            (for i = 0 to r-1 do
              (for j = 0 to c-1 do
                let ele_ptr_ptr = (L.build_gep mat [|L.const_int i32_t (i*c+j)|]
"element_ptr_ptr" builder) in
                let ele = L.build_load ele_ptr_ptr "element_ptr" builder in
                (if ((s1<=i) && (i<=t1) && (s2<=j) && (j<=t2)) then (
                  let res_ptr_ptr = L.build_gep res [|L.const_int i32_t (!pointer)|]
"res_ptr_ptr" builder in
                  ignore(L.build_store ele res_ptr_ptr builder);
                  pointer := !pointer+1;
                ))
              done);
            done);res_mat)

          | (e1,(_,SRange(rs2,rt2))) ->
            (let (r,c) = find_size_inmap s in
            let (rs2,rt2) = (match e2 with (_,SRange(rs2,rt2)) -> (rs2,rt2)) in
            let s2 = match rs2 with SBeg -> 0 | SEnd -> c-1 | SInd(s2) -> s2 in
            let t2 = match rt2 with SBeg -> 0 | SEnd -> c-1 | SInd(t2) -> t2 in

            let ptr = L.build_load (lookup s) s builder in
            let mat = L.build_load (L.build_struct_gep ptr 0 "m_mat" builder) "mat" builder
in

            let res_mat = build_default_mat (1,(t2-s2+1)) builder in
            let res = L.build_load (L.build_struct_gep res_mat 0 "m_mat" builder) "mat"
builder in

            let pointer = ref 0 in

            let i = expr builder e1 in

            (for j = 0 to c-1 do
              let index = L.build_add (L.build_mul i (L.const_int i32_t c) "mul_tmp" builder)
(L.const_int i32_t j) "add_tmp" builder in
              let ele_ptr_ptr = (L.build_gep mat [|index|] "element_ptr_ptr" builder) in
              let ele = L.build_load ele_ptr_ptr "element_ptr" builder in

              (if ((s2<=j) && (j<=t2)) then (
                let res_ptr_ptr = L.build_gep res [|L.const_int i32_t (!pointer)|]
"res_ptr_ptr" builder in
                ignore(L.build_store ele res_ptr_ptr builder);
                pointer := !pointer+1;
              ))
            done);res_mat)

          | ((_,SRange(rs1,rt1)),e2) ->
            (let (r,c) = find_size_inmap s in
            let (rs1,rt1) = (match e1 with (_,SRange(rs1,rt1)) -> (rs1,rt1)) in
            let s1 = match rs1 with SBeg -> 0 | SEnd -> r-1 | SInd(s1) -> s1 in
            let t1 = match rt1 with SBeg -> 0 | SEnd -> r-1 | SInd(t1) -> t1 in

            let ptr = L.build_load (lookup s) s builder in
            let mat = L.build_load (L.build_struct_gep ptr 0 "m_mat" builder) "mat" builder
```

```
in

           let res_mat = build_default_mat ((t1-s1+1),1) builder in
           let res = L.build_load (L.build_struct_gep res_mat 0 "m_mat" builder) "mat"
builder in

           let pointer = ref 0 in

           let j = expr builder e2 in

           (for i = 0 to r-1 do
             let index = L.build_add (L.build_mul (L.const_int i32_t i) (L.const_int i32_t
c) "mul_tmp" builder) j "add_tmp" builder in
             let ele_ptr_ptr = (L.build_gep mat [|index|] "element_ptr_ptr" builder) in
             let ele = L.build_load ele_ptr_ptr "element_ptr" builder in

             (if ((s1<=i) && (i<=t1)) then (
               let res_ptr_ptr = L.build_gep res [|L.const_int i32_t (!pointer)|]
"res_ptr_ptr" builder in
               ignore(L.build_store ele res_ptr_ptr builder);
               pointer := !pointer+1;
             ))
           done);res_mat)
       )

     | SEmpty     -> L.const_int i32_t 0
     | SVar s     -> L.build_load (lookup s) s builder

     | SStruaccess(vname, member) -> let stn = StringMap.find vname stru_name in
       let (sdef,sdecl) = StringMap.find stn struct_decls in
       let mem_idx =
         let rec find_idx m mlist = match mlist with
             [] -> raise (Failure ("unrecognized struct member " ^ m))
           | Primdecl(_, nm) :: tl -> if m = nm then 0 else 1 + find_idx m tl
         in find_idx member sdecl.sstvar
       in let str_ptr = L.build_load (lookup vname) vname builder
       in L.build_load (L.build_struct_gep str_ptr mem_idx (stn ^ member) builder) (vname ^
member) builder

     | SAssign (s, e) ->
       let e' = expr builder e in
         (match s with
           | (_,SVar(s1)) -> ignore(L.build_store e' (lookup s1) builder); e'
           | (_,SStruaccess(vname, member)) ->
               let mem_ptr =
               let stn = StringMap.find vname stru_name in
               let (sdef,sdecl) = StringMap.find stn struct_decls in
               let mem_idx =
                 let rec find_idx m mlist = match mlist with
                     [] -> raise (Failure ("unrecognized struct member " ^ m))
```

```
                  | Primdecl(_, nm) :: tl -> if m = nm then 0 else 1 + find_idx m tl
                in find_idx member sdecl.sstvar
              in let str_ptr = L.build_load (lookup vname) vname builder
              in L.build_struct_gep str_ptr mem_idx (stn ^ member) builder
              in ignore(L.build_store e' mem_ptr builder); e'
          | (_,SMataccess (s,e1,e2)) ->
               let idx =
                 let e1' = expr builder e1 and e2' = expr builder e2 in
                 let ptr = L.build_load (lookup s) s builder in
                 let mat = L.build_load (L.build_struct_gep ptr 0 "m_mat" builder) "mat"
builder in
                 let r = L.build_load (L.build_struct_gep ptr 1 "m_r" builder) "r_mat"
builder in
                 let c = L.build_load (L.build_struct_gep ptr 2 "m_c" builder) "c_mat"
builder in
                 let index = L.build_add e2' (L.build_mul e1' c "tmp" builder) "index"
builder in
                 let (fdef, _) = StringMap.find "index_check" function_decls in
                 ignore(L.build_call fdef [| e1'; r |] "" builder);
                 ignore(L.build_call fdef [| e2'; c |] "" builder);
                 L.build_gep mat [|index|] "element_ptr_ptr" builder
                in
                ignore(L.build_store e' idx builder); e'
          | _ -> raise (Failure "Assign Failiure!"))

    | SBinop (e1, op, e2) ->
    (let c1 = expr builder e1 in
    let c2 = expr builder e2 in

    let check1 = is_matrix c1 in
    let check2 = is_matrix c2 in
    match (check1,check2) with
      | (false,false) ->
        (match e1 with
          | (A.Float,_) ->
            let e1' = expr builder e1
            and e2' = expr builder e2 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.Eq      -> L.build_fcmp L.Fcmp.Oeq
            | A.Neq     -> L.build_fcmp L.Fcmp.One
            | A.Less    -> L.build_fcmp L.Fcmp.Olt
            | A.Leq     -> L.build_fcmp L.Fcmp.Ole
            | A.Greater -> L.build_fcmp L.Fcmp.Ogt
            | A.Geq     -> L.build_fcmp L.Fcmp.Oge
            | A.And | A.Or ->
                raise (Failure "internal error: semant should have rejected and/or on
float")
             ) e1' e2' "tmp" builder
           | _ ->
             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.And     -> L.build_and
             | A.Or      -> L.build_or
             | A.Eq      -> 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)
       | (true, false) ->
          let (r1,c1) = lookup_size e1 in
          let e1' = expr builder e1 and e2' = expr builder e2 in
          (match op with
           | A.Mult -> mat_float_operation e1' r1 c1 e2' builder
           | A.Div -> mat_float_operation e1' r1 c1 (L.build_fdiv (L.const_float float_t
1.0) e2' "tmp" builder) builder
          )
       | (false, true) ->
          let (r2,c2) = lookup_size e2 in
          let e1' = expr builder e1 and e2' = expr builder e2 in
          (match op with
           | A.Mult -> mat_float_operation e2' r2 c2 e1' builder)
       | (true,true) ->
         let (r1,c1) = lookup_size e1 in
         let (r2,c2) = lookup_size e2 in
         let e1' = expr builder e1 and e2' = expr builder e2 in
         (match op with
             A.Add -> mat_mat_operation e1' e2' r1 c1 r2 c2 0 "yes" builder
           | A.Sub -> mat_mat_operation e1' e2' r1 c1 r2 c2 1 "yes" builder
           | A.Mult -> mat_mat_operation e1' e2' r1 c1 r2 c2 4 "not" builder
           | A.Elemult -> mat_mat_operation e1' e2' r1 c1 r2 c2 2 "yes" builder
           | A.Elediv -> mat_mat_operation e1' e2' r1 c1 r2 c2 3 "yes" builder
         )
     )

     | SUop(op, ((t, _) as e)) ->
       let e' = expr builder e in
       (match op with
         A.Nega when t = A.Float -> L.build_fneg
       | A.Nega                  -> L.build_neg
```

```
        | A.Not                   -> L.build_not) e' "tmp" builder

      | SCall ("print", [e]) ->
         L.build_call printf_func [| int_format_str ; (expr builder e) |]
        "printf" builder

      (*Add print functions, other built-in functions*)
      | SCall ("printStr", [e]) ->
        L.build_call printf_func [| string_format_str ; (expr builder e) |]
        "printf" builder

      | SCall ("printFloat",[e]) ->
        L.build_call printf_func [| float_format_str ; (expr builder e) |]
        "printf" builder


      | SCall ("imread", [s;e;e2;e3]) ->
        let path = expr builder s in
        let str_ptr = expr builder e in
        let r = match e2 with (_,SIntlit(r)) -> r in
        let c = match e3 with (_,SIntlit(c)) -> c in

        let mat1_ptr = L.build_load (L.build_struct_gep str_ptr 0 ("R") builder) ("sR")
builder in
        let mat2_ptr = L.build_load (L.build_struct_gep str_ptr 1 ("G") builder) ("sG")
builder in
        let mat3_ptr = L.build_load (L.build_struct_gep str_ptr 2 ("B") builder) ("sB")
builder in
        let mat1 = L.build_load (L.build_struct_gep (mat1_ptr) 0 "m_mat" builder) "mat_mat"
builder in
        let mat2 = L.build_load (L.build_struct_gep (mat2_ptr) 0 "m_mat" builder) "mat_mat"
builder in
        let mat3 = L.build_load (L.build_struct_gep (mat3_ptr) 0 "m_mat" builder) "mat_mat"
builder in
        ignore (L.build_call load_cpp_func [| path;mat1;mat2;mat3 |] "" builder );str_ptr


      | SCall ("imwrite", [s;e]) ->
        let path = expr builder s in
        let str_ptr = expr builder e in
        let mat1_ptr = L.build_load (L.build_struct_gep str_ptr 0 ("R") builder) ("sR")
builder in
        let mat2_ptr = L.build_load (L.build_struct_gep str_ptr 1 ("G") builder) ("sG")
builder in
        let mat3_ptr = L.build_load (L.build_struct_gep str_ptr 2 ("B") builder) ("sB")
builder in
        let mat1 = L.build_load (L.build_struct_gep (mat1_ptr) 0 "m_mat" builder) "mat_mat"
builder in
        let mat2 = L.build_load (L.build_struct_gep (mat2_ptr) 0 "m_mat" builder) "mat_mat"
builder in
```

```
       let mat3 = L.build_load (L.build_struct_gep (mat3_ptr) 0 "m_mat" builder) "mat_mat"
builder in
       let r = L.build_load (L.build_struct_gep (mat1_ptr) 1 "m_mat" builder) "mat_mat"
builder in
       let c = L.build_load (L.build_struct_gep (mat1_ptr) 2 "m_mat" builder) "mat_mat"
builder in

       ignore(L.build_call save_cpp_func [| path;mat1;mat2;mat3;r;c |] "" builder);mat1

     | SCall("abort",[]) ->
       ignore(L.build_call printf_func [| string_format_str ; L.build_global_stringptr
"Error: Matrix index out of bound" "tmp" builder|] "printf" builder);
       L.build_call abort_func [| |] "" builder

     | SCall ("height",[e]) ->
       let r = L.build_load (L.build_struct_gep (expr builder e) 1 "m_r" builder) "r_mat"
builder in
       r

     | SCall ("width", [e]) ->
       let c = L.build_load (L.build_struct_gep (expr builder e) 2 "m_c" builder) "c_mat"
builder in
       c

     | SCall ("sum", [e]) ->
       let (r,c) = lookup_size e in
       let mat = L.build_load (L.build_struct_gep (expr builder e) 0 "m_mat" builder)
"mat_mat" builder in

       let sum = L.build_alloca float_t "sumOfEle" builder in
       (for x=0 to r*c-1 do
         let ele_ptr_ptr = (L.build_gep mat [|L.const_int i32_t x|] "element_ptr_ptr"
builder) in
         let ele = L.build_load ele_ptr_ptr "element_ptr" builder in
         let tmp_sum = L.build_fadd (L.build_load sum "addsum" builder) ele "tmp_sum"
builder in
           ignore(L.build_store tmp_sum sum builder);
       done);
       L.build_load sum "addsum" builder

     | SCall ("mean", [e]) ->
       let (r,c) = lookup_size e in
       let mat = L.build_load (L.build_struct_gep (expr builder e) 0 "m_mat" builder)
"mat_mat" builder in

       let sum = L.build_alloca float_t "sumOfEle" builder in
       (for x=0 to r*c-1 do
         let ele_ptr_ptr = (L.build_gep mat [|L.const_int i32_t x|] "element_ptr_ptr"
builder) in
         let ele = L.build_load ele_ptr_ptr "element_ptr" builder in
         let tmp_sum = L.build_fadd (L.build_load sum "addsum" builder) ele "tmp_sum"
builder in
           ignore(L.build_store tmp_sum sum builder);
       done);
       L.build_fdiv (L.build_load sum "addsum" builder) (L.const_float float_t 4.0)
"mean_sum" builder

     | SCall ("trans", [e]) ->
       let (r,c) = lookup_size e in
       let mat = L.build_load (L.build_struct_gep (expr builder e) 0 "m_mat" builder)
"mat_mat" builder in
       let res_mat = build_default_mat (c,r) builder in
       let res = L.build_load (L.build_struct_gep res_mat 0 "m_mat" builder) "mat" builder
in
       ignore (L.build_call trans_cpp_func [| mat;res;(L.const_int i32_t r);(L.const_int
i32_t c) |] "" builder );
       res_mat

     | SCall ("cov_openCV", [k;e1;e2;sd]) ->
       let fromPath = expr builder e1 in
       let toPath = expr builder e2 in
       let d = expr builder sd in
       let str_ptr = expr builder k in
       let kernel = L.build_load (L.build_struct_gep (str_ptr) 0 "k_mat" builder)
"mat_mat_k" builder in
       L.build_call filter_cpp_func [| kernel; fromPath; toPath;d |] "" builder

     | SCall ("cov", [e1;e2]) ->
       let (r1,c1) = lookup_size e1 in
       let (r2,c2) = lookup_size e2 in
       let d = (r2-1)/2 in
       let mat1 = L.build_load (L.build_struct_gep (expr builder e1) 0 "m_mat" builder)
"mat_mat" builder in
       let mat2 = L.build_load (L.build_struct_gep (expr builder e2) 0 "m_mat" builder)
"mat_mat" builder in
       let res_mat = build_default_mat (r1,c1) builder in
       let res = L.build_load (L.build_struct_gep res_mat 0 "m_mat" builder) "mat" builder
in

       let tmp_mat = build_default_mat (r1+2*d,c1+2*d) builder in
       let tmp = L.build_load (L.build_struct_gep tmp_mat 0 "m_mat" builder) "mat" builder
in

       (for i=0 to (r1+d) do
         (for j=0 to (c1+d) do
           let ele =
             (if (((i-d)<0) || ((j-d)<0) || ((i+d)>(r1+d)) || ((j+d)>(c1+d))) then
(L.const_float float_t 0.0)
             else (let ele_ptr_ptr = (L.build_gep mat1 [|L.const_int i32_t ((i-d)*c1+j-d)|]
"element_ptr_ptr" builder) in
```

```
                L.build_load ele_ptr_ptr "element_ptr" builder))
            in
            let tmp_ptr_ptr = L.build_gep tmp [|L.const_int i32_t (i*(c1+2*d)+j)|]
"tmp_ptr_ptr" builder in
            ignore(L.build_store ele tmp_ptr_ptr builder)
          done);
        done);

        (for i=d to r1 do
          (for j=d to c1 do
            let res_val_tmp = L.build_alloca float_t "sumOfEle" builder in
            let ptr = ref 0 in
            (for x=(i-d) to (i+d) do
              (for y=(j-d) to (j+d) do
                let mat2_ele_ptr_ptr = (L.build_gep mat2 [|L.const_int i32_t !ptr|]
"mat2_ele_ptr_ptr" builder) in
                let mat2_ele = L.build_load mat2_ele_ptr_ptr "mat2_ele_ptr" builder in
                let tmp_ele_ptr_ptr = (L.build_gep tmp [|L.const_int i32_t (x*(c1+2*d)+y)|]
"mat2_ele_ptr_ptr" builder) in
                let tmp_ele = L.build_load tmp_ele_ptr_ptr "tmp_ele_ptr" builder in

                let tmp_res = L.build_fadd (L.build_fmul mat2_ele tmp_ele "tmp_res" builder)
                                           (L.build_load res_val_tmp "addres" builder)
"new_res" builder in
                                           ignore(L.build_store tmp_res res_val_tmp builder);
                ptr := !ptr+1;
              done);
            done);
            ignore(L.build_store (L.build_load res_val_tmp "res" builder)
                  (L.build_gep res [|L.const_int i32_t ((i-d)*(c1)+j-d)|] "tmp_ptr_ptr"
builder) builder)
          done);
        done); res_mat

      | 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.sftyp 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.sftyp 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

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

```
     | SInitial (typ, name, e) -> (match e with
                                       (_, SEmpty) -> builder
                                     | _ -> let e' = expr builder e in
                                            (ignore(L.build_store e' (lookup name) builder);
builder))
     | SDefaultmat (name, r, c) -> (L.build_store (build_default_mat (r,c) builder) (lookup
name) builder);builder
     | SIniStrucct (var, strucname, mems) ->
       let (sdef,_) = StringMap.find strucname struct_decls in
       let llmems = List.rev (List.map (expr builder) (List.rev mems)) in
       let str_ptr = L.build_malloc sdef (var ^ "_malloc") builder in
       let build_struct i llmem =
         ignore(L.build_store llmem (L.build_struct_gep str_ptr i (var ^ "_" ^string_of_int
i) builder) builder); i+1
       in ignore(List.fold_left build_struct 0 llmems);
       ignore(L.build_store str_ptr (lookup var) 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.sftyp 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
```

8.7 io.cpp

```
#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/opencv.hpp>
#include "opencv2/objdetect/objdetect.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"

#include <stdio.h>
#include <iostream>
#include <string>
#include <cmath>
```

```
using namespace cv;

using namespace std;
extern "C" void load_cpp(char imageName[],double* mat1,double* mat2,double* mat3)
{
    Mat img = imread(imageName,CV_LOAD_IMAGE_COLOR);
    unsigned char* input = (unsigned char*)(img.data);
    double r,g,b;
    for(int i = 0;i < img.rows;i++){
        for(int j = 0;j < img.cols;j++){
            int k = img.step * i + j*img.channels();
            int s = i*img.cols + j;
            b = input[k] ;
            mat1[s]=b;
            g = input[k + 1];
            mat2[s]=g;
            r = input[k + 2];
            mat3[s]=r;
        }
    }

    return;
}

extern "C" void save_cpp(char fileName[],double* mat1,double* mat2,double* mat3,int r, int
c)
{
    int height = r;
    int width = c;
    double* data = new double[3*width*height];

    int ind = 0;
    int i = 0;
    while(i<=3*width*height){
        data[i]=mat1[ind];
        data[i+1]=mat2[ind];
        data[i+2]=mat3[ind];
        i=i+3;
        ind++;
    }

    cout << "f" <<data[0] <<endl;
    cout << "f" <<data[1] <<endl;
    cout << "f" <<data[2] <<endl;

    cout << "f" <<data[3] <<endl;
    cout << "f" <<data[4] <<endl;
    cout << "f" <<data[5] <<endl;
    cout << "f" <<data[6] <<endl;
    cout << "f" <<data[7] <<endl;
    cout << "f" <<data[8] <<endl;

    Mat image = cv::Mat(height, width, CV_64FC3, data);
    imwrite(fileName,image);
    cout << "output width: " << width << endl;
    cout << "output height: " << height << endl;
    return;
}

extern "C" void filter_cpp (double* kernel_d, char inputName[],char outputName[],int d)
{
    Mat dst;
    int ddepth = -1;
    double delta = 0.0;
    Point anchor = Point( -1, -1);

    Mat image = imread(inputName,CV_LOAD_IMAGE_COLOR);

    double k[d][d];
    for(int i=0;i<d;++i){
        for(int j=0;j<d;++j){
            k[i][j] = kernel_d[i*d+j];
        }
    }

    Mat kernel = Mat(d,d,CV_64F,k);

    filter2D(image, dst, ddepth , kernel, anchor, delta, BORDER_DEFAULT);
    imwrite(outputName,dst);

    return;
}

extern "C" void iter1mat_cpp (double* indata, double* outdata, double k, int len)
{
    for(int i=0;i<len;++i){
        outdata[i] = indata[i]*k;
    }
    return;
}

extern "C" void trans_cpp (double* indata, double* outdata, int r, int c)
{
    for(int i=0;i<r;++i){
        for(int j=0;j<c;++j){
            outdata[j*r+i] = indata[i*c+j];
        }
    }
    return;
}

extern "C" void iter2mat_cpp (double* indata1, double* indata2, double* outdata, int mode,
int len)
{
    switch (mode){
        case 0:
        for(int i=0;i<len;++i){
            outdata[i] = indata1[i] + indata2[i];
        }
        break;
        case 1:
        for(int i=0;i<len;++i){
            outdata[i] = indata1[i] - indata2[i];
        }
        break;
        case 2:
        for(int i=0;i<len;++i){
            outdata[i] = indata1[i] * indata2[i];
        }
        break;
        case 3:
        for(int i=0;i<len;++i){
            outdata[i] = indata1[i] / indata2[i];
        }
        break;
    }
    return;
}

extern "C" void matmul_cpp (double* indata1, double* indata2, double* outdata, int r1, int
c1, int r2, int c2)
{
    for(int i=0;i<r1;++i){
        for(int j=0;j<c2;++j){
            int idx = i*c2+j;
            float tmp_sum = 0.0;
            int x = i * c1;
            int y = j;
            while (y < r2*c2){
                tmp_sum += indata1[x]*indata2[y];
                x = x + 1;
                y = y + c2;
            }
            outdata[idx] = tmp_sum;
        }
    }
    return;
}
```

8.8 compile.sh

8.9 tests

8.9.1 fail-assign1.mmm

```
func void main(){
    int i;
    bool b;
    i = false;
}
```

8.9.2 fail-assign2.mmm

```
func int main(){
    int i = 1;
    float f;

    f = 1;
}
```

8.9.3 fail-assign3.mmm

```
func int main(){
    int i;
    string s;

    s = 1; /* should fail because cannot assign string to int value */
}
```

8.9.4 fail-assign4.mmm

```
func int main(){
    int a = 1;
    matrix b[4,5];

    b = a;  /* should fail because assign int to matrix */
}
```

8.9.5 fail-declare.mmm

```
func int main(){
    int a = 3;
    int a = 5;
    return 0;
}
```

8.9.6 fail-for1.mmm

```
func void main(){
    for(i=0;i=5;i++){
        print("aaa");   /* this should fail because not binary operation */
    }
}
```

8.9.7 fail-func1.mmm

```
func void print(){
    string a = "aaaa";
    print(a);    /* should fail because cannot have function named print */
}

func void main(){
    print();
}
```

8.9.8 fail-func2.mmm

```
func matrix height(matrix m){  /* should fail because cannot have function named height */
    m = m * m;
    return m;
}

func void main(){
    matrix m = [1.1,1.0;2.2,2.3];
    height(m);
}
```

8.9.9 fail-func3.mmm

```
func matrix Foo(){
    matrix m = [1.1,1.0,1.2;2.2,2.3,2.4;3.1,3.2,3.3];
    return m;
}

func void main(){
    matrix a[3, 3];
    a = Bar();   /* this should fail because unrecognized func */
}
```

8.9.10 fail-if1.mmm

```
func void main(){
    if(true){ }
    if(1<2){ }
    if(false){ } else { }
    if(666){ }          /* this should fail because 42 is not bool expr*/
}
```

8.9.11 fail-matrix-add.mmm

```
func int main() {
    matrix a = [1.1,2.3];
    matrix b = [1.1];
    a+b;
    return 0;
}
```

8.9.12 fail-matrix-assign.mmm

```
func int main() {
    matrix a = [0.0,0.1;1.0,1.1;2.0,2.1];


    matrix res[2,3];
    res = a;
    return 0;
}
```

8.9.13 fail-matrix-elemul.mmm

```
func int main() {
    matrix a = [1.1,2.3];
    matrix b = [1.1];
    a.*b;
    return 0;
}
```

8.9.14 fail-matrix-mul.mmm

```
func int main() {
    matrix a = [1.1,2.3];
    matrix b = [1.1];
    a*b;
    return 0;
}
```

8.9.15 fail-matrix1.mmm

```
func matrix Foo(){
    matrix m = [1.1,1.0,1.2;2.2,2.3,2.4;3.1,3.2,3.3];
    int a = mean(123);  /* this should fail because mean input a int */
    return a;
}

func void main(){
    matrix b = Foo();
}
```

8.9.16 fail-matrix2.mmm

```
func matrix Foo(){
    matrix m = [1.1,1.0,1.2;2.2,2.3,2.4;3.1,3.2,3.3];
    return m;
}

func void main(){
    matrix b[3,3];
    b = Foo();
    int c = b[1.1][1.2];
}
```

8.9.17 fail-struct1.mmm

```
struct Image {
    int BW;
    int R;
    int B
}

func int main()
{
    struct K = Image(1,2,3);
    int a = K.C;
    print(1);
    return 1;
}
```

8.9.18 fail-struct2.mmm

```
struct Image {
    matrix BW;
    int R;
    int B
}

func int main()
{
    struct K = Image(1,2,3);
    int a = K.R;
    print(1);
    return 1;
}
```

8.9.19 fail-struct3.mmm

```
struct Image {
    int BW;
    int R;
    int B
}

func int main()
{
    struct K = Image(1,2,3);
    float a = K.R;
    print(1);
    return 1;
}
```

8.9.20 test-binop.mmm

```
func int main() {

    print(3+6);
    print(3-6);
    print(3*6);
    print(6/3);
    return 0;
}
```

8.9.21 test-bool1.mmm

```
func int main(){
    if(true){
        print(100);
    }
    return 0;
}
```

8.9.22 test-bool2.mmm

```
func int main(){
    bool a = false;
    if(a == false){
        print(100);
    }
    return 0;
}
```

8.9.23 test-comment.mmm

```
func int main(){
    // test comment
    int a = 3;
    print(a);
    return 0;
}
```

8.9.24 test-comment2.mmm

```
func int main(){
    /* this
    is
    a
    comment
    */

    int a = 3;
    print(a);
    return 0;
}
```

8.9.25 test-comparator.mmm

```
func int main() {
    if (1 != 2) {
        print(1);
    }
    else {
        print(0);
    }
    if (1 == 1) {
        print(1);
    }
    else {
        print(0);
    }
    if (1 < 2) {
        print(1);
    }
    else {
        print(0);
    }
    if (1 > 2) {
        print(0);
    }
    else {
        print(1);
    }
    if (1 <= 1) {
        print(1);
    }
    else {
        print(0);
    }
    if (1 >= 0) {
        print(1);
    }
    else {
        print(0);
    }
    return 0;
}
```

8.9.26 test-demo.mmm

```
func int main()
{
    // assignment
    matrix m1 = [1.1,2.2,3.3;4.4,5.5,6.6;7.7,8.8,9.9];
    matrix m2[3,3];
    m2 = m1;

    // indexing
    float x = m1[2][0];

    // slicing
    int a = 1;

    matrix sliced[2,2];
    sliced = m1[0:1][0:1];

    matrix sliced2[1,2];
    sliced2 = m1[1][0:1];

    matrix sliced3[1,2];
    sliced3 = m1[a][0:1];

    matrix sliced4[2,1];
    sliced4 = m1[0:1][a];

    return 0;
}
```

8.9.27 test-demo2.mmm

```
func int main()
{
    matrix m1 = [1.1,2.2,3.3;4.4,5.5,6.6];
    matrix m2 = [1.2,2.3,3.4;4.5,5.6,6.7];
    matrix m3 = [1.2,2.3;3.4,4.5;5.6,6.7];
    matrix res[2,3];
    res = m1+m2;
    res = m1./m2;
    res = m1.*m2;
    res = m1-m2;

    matrix res2[2,2];
    res2 = m1*m3;

    matrix a[3,2];
    a = trans(m1);

    float m = mean(m1);
    float s = sum(m1);
    int w = width(m1);
    int h = height(m1);

    return 0;
}
```

8.9.28 test-floatAssign1.mmm

```
func int main()
{
    float a = 6.7;
    printFloat(a);
    return 0;

}
```

8.9.29 test-for1.mmm

```
func int main(){
    int a = 0;
    for (a = 0; a < 3; a = a + 1) {
        print(2);
    }
    return 0;
}
```

8.9.30 test-func-matrix.mmm

```
func int main() {
    matrix a = [1.1,2.2;3.3,4.4;5.5,6.6];
    float res = f(a);
    printFloat(res);
    return 0;
}
func float f(matrix m){

 return m[1][1];
 }
```

8.9.31 test-func-matrix2.mmm

```
func int main() {
    matrix x = [0.0,0.1;1.0,1.1;2.0,2.1];


    matrix res[3,2];
    res = f(x);
    printFloat(res[2][1]);
    return 0;
}
func matrix f(matrix m){
    matrix a = [0.0,0.1;1.0,1.1;2.0,2.1];
    return a;
}
```

8.9.32 test-func-matrix3.mmm

```
func int main() {
    matrix x = [0.0,0.1;1.0,1.1;2.0,2.1];


    matrix res[3,2];
    printFloat(res[0][0]);
    return 0;
}
```

8.9.33 test-functionCall.mmm

```
func int main(){
    int a = 1;
    f(a);
    return 0;
}
func void f(int x) {
    print(x);
}
```

8.9.34 test-if1.mmm

```
func int main(){
    bool a = true;
    if(a) {
        print(1);
    }
    else {
        print(0);
    }
    bool b = false;
    if(b) {
        print(0);
    }
    else {
        print(1);
    }
}
```

8.9.35 test-logic.mmm

func int main() {

```
    if (true && true) {
        print(1);
    }
    if (false && true) {
        print(0);
    }
    if (true || false) {
        print(1);
    }
    if (!false) {
        print(0);
    }
    return 0;
}
```

8.9.36 test-matrix-cov.mmm

```
func int main()
{
    matrix m1 = [1.0,2.0,3.0,3.1;4.0,5.0,6.0,6.1;7.0,8.0,9.0,9.1];
    matrix m2 = [1.1,2.2,3.3;4.4,5.5,6.6;7.7,8.8,9.9];
    matrix a[3,4];
    a = cov(m1,m2);
    printFloat(a[2][2]);
    return 0;
}
```

8.9.37 test-matrix-default.mmm

```
func int main()
{
     matrix m1 = [1.1,2.2,3.3;4.4,5.5,6.6];
     //matrix res[2,1];
     matrix res = m1[0:1][1];
     float a = res[0][0];
     printFloat(a);
     return 0;
}
```

8.9.38 test-matrix-height.mmm

```
func int main()
{
    matrix m = [1.1,2.2;3.3,4.4;5.5,6.6];
    int a;
    a = height(m);
    print(a);
    return 0;
}
```

8.9.39 test-matrix-mean.mmm

```
func int main()
{
    matrix m = [1.1,2.2;3.3,4.4];
    float a;
    a = mean(m);
    printFloat(a);
    return 0;
}
```

8.9.40 test-matrix-op-add.mmm

```
func int main()
{
    matrix m1 = [1.1,2.2;3.3,4.4];
    matrix m2 = [1.1,2.2;3.3,4.4];
    matrix res[2,2];
    res = m1+m2;
    float a = res[1][1];
    printFloat(a);
    return 0;
}
```

8.9.41 test-matrix-op-div.mmm

```
func int main()
{
    matrix m1 = [1.1,2.2,3.3;4.4,5.5,6.6];
    matrix m2 = [1.2,2.3,3.4;4.5,5.6,3.3];
    matrix res[2,3];
    res = m1./m2;
    float a = res[1][2];
    printFloat(a);
    return 0;
}
```

8.9.42 test-matrix-op-elemul.mmm

```
func int main()
{
    matrix m1 = [1.1,2.2,3.3;4.4,5.5,6.6];
    matrix m2 = [1.2,2.3,3.4;4.5,5.6,3.3];
    matrix res[2,3];
    res = m1.*m2;
    float a = res[1][2];
    printFloat(a);
    return 0;
}
```

8.9.43 test-matrix-op-mul.mmm

```
func int main()
{
    matrix m1 = [1.1,2.2,3.3;4.4,5.5,6.6];
    matrix m2 = [1.2,2.3;3.4,4.5;5.6,6.7];
    matrix res[2,2];
    res = m1*m2;
    float a = res[0][0];
    printFloat(a);
    return 0;
}
```

8.9.44 test-matrix-op-sub.mmm

```
func int main()
{
    matrix m1 = [1.1,2.2,3.3;4.4,5.5,6.6];
    matrix m2 = [1.2,2.3,3.4;4.5,5.6,6.7];
    matrix res[2,3];
    res = m1-m2;
    float a = res[1][2];
    printFloat(a);
    return 0;
}
```

8.9.45 test-matrix-slice.mmm

```
func int main()
{
    matrix m1 = [1.1,2.2,3.3;4.4,5.5,6.6];
    matrix res[2,1];
    res = m1[0:1][1];
    float a = res[0][0];
    printFloat(a);
    return 0;
}
```

8.9.46 test-matrix-sum.mmm

```
func int main()
{
    matrix m = [1.1,2.2;3.3,4.4];
    float a;
    a = sum(m);
    printFloat(a);
    return 0;
}
```

8.9.47 test-matrix-trans.mmm

```
func int main()
{
    matrix m = [1.1,2.2,3.3;4.4,5.5,6.6];
    matrix a[3,2];
    a = trans(m);
    printFloat(a[2][0]);
    return 0;
}
```

8.9.48 test-matrix-width.mmm

```
func int main()
{
    matrix m = [1.1,2.2;3.3,4.4;5.5,6.6];
    int a;
    a = width(m);
    print(a);
    return 0;
}
```

8.9.49 test-matrix.mmm

```
func int main()
```

```
{
    matrix m = [1.1,2.2;3.3,4.4];
    print(1);
    return 0;
}
```

8.9.50 test-matrix1.mmm

```
func int main() {
    matrix m = [1.1,2.2];
    return 0;
}
```

8.9.51 test-matrix2.mmm

```
func int main()
{
    matrix m = [1.1,2.2;3.3,4.4;5.5,6.6];
    float x = m[2][0];
    printFloat(x);
    return 0;
}
```

8.9.52 test-printFloat1.mmm

```
func int main()
{
    printFloat(6.66);
    return 0;

}
```

8.9.53 test-printInt.mmm

```
func int main()
{
    print(66666);
    return 0;

}
```

8.9.54 test-printInt2.mmm

```
func int main()
{
    int a;
    a = 11111;
    print(a);
    return 0;
}
```

8.9.55 test-printInt3.mmm

```
func void foo(int a)
{
    print(a + 3);
}
func int main() {
    foo(40);
    return 0;
}
```

8.9.56 test-printStr1.mmm

```
func int main(){

    printStr("hello world");
    return 0;
}
```

8.9.57 test-return1.mmm

```
func int main() {
    print(f());
    return 0;
}
func int f() {
    return 3;
}
```

8.9.58 test-return2.mmm

```
func int main() {
    f();
    return 0;
}
func void f() {
    print(3);
}
```

8.9.59 test-stringAssign1.mmm

```
func int main(){
    string a = "hello";
    printStr(a);
    return 0;
}
```

8.9.60 test-struct1.mmm

```
struct Image {
    int BW;
    float R;
    int B
}

func int main()
{
    struct K = Image(1,2.5,3);
    float a = K.R;
    printFloat(a);
    return 0;
}
```

8.9.61 test-struct2.mmm

```
struct Image {
    int BW;
    float R;
    matrix X
}

func int main()
{
    struct K = Image(1,2.5,[1.1,2.2;3.3,4.4;5.5,6.6]);
    matrix a[3,2];
    a = K.X;
    float s = a[2][1];
    printFloat(s);
    return 0;
}
```

8.9.62 test-struct3.mmm

```
struct Image {
    bool BW;
    float R;
    matrix X
}

func float foo(<Image> T, float c)
{
    matrix a[3,2];
    a = T.X;
    float s = a[0][1] + c;
    return s;
}

func int main()
{
    struct K = Image(true,2.5,[1.1,2.2;3.3,4.4;5.5,6.6]);
    float m = foo(K, 1.3);
    printFloat(m);
    return 0;
}
```

8.9.63 test-struct4.mmm

```
struct Image {
    int BW;
    float R;
    int B
}

func int main()
{
    struct K = Image(1,2.5,3);
    K.R = 6.6;
    float a = K.R;
    printFloat(a);
    return 0;
}
```

8.9.64 test-while1.mmm

```
func int main(){
    int a = 5;
    while (a > 0) {
        print(a);
        a = a - 1;
    }
    return 0;
}
```