

# MatCV - Project Report

Abhishek Walia (aw3011),  
Anuraag Advani (ada2161),  
Shardendu Gautam (sg3391)

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Motivation . . . . .	3
1.2	Description . . . . .	3
1.3	Features . . . . .	3
<b>2</b>	<b>Tutorial</b>	<b>3</b>
2.1	Running the compiler . . . . .	3
2.2	Matrix Hello World . . . . .	4
<b>3</b>	<b>Language Reference Manual</b>	<b>4</b>
3.1	Data Types . . . . .	4
3.2	Operators . . . . .	5
3.2.1	Matrix Operations . . . . .	6
3.2.2	Operator Precedence . . . . .	6
3.3	Comments . . . . .	6
3.4	Keywords . . . . .	6
3.5	Identifiers . . . . .	7
3.6	Library Functions . . . . .	8
3.7	Matrix Initialization . . . . .	8
3.8	Functions . . . . .	8
3.9	Structure . . . . .	8
<b>4</b>	<b>Project Plan</b>	<b>9</b>
4.1	Process . . . . .	9
4.2	Team Roles . . . . .	9
4.3	Programming Style Guide . . . . .	9
4.4	Project Timeline . . . . .	10
4.5	Challenges . . . . .	10
4.6	Development Environment . . . . .	10
<b>5</b>	<b>Architecture</b>	<b>10</b>
5.1	Scanner . . . . .	11
5.2	Parser . . . . .	11
5.3	Semantic Checker . . . . .	12
5.4	MatCV Library . . . . .	13
5.5	Code Generator . . . . .	13
<b>6</b>	<b>Testing</b>	<b>13</b>
6.1	Test Plan . . . . .	13
6.2	Automation . . . . .	13
6.3	Demo Code - Source Code to Target Language . . . . .	15
6.4	Program 1 . . . . .	15
6.5	Program 2 . . . . .	23

<b>7</b>	<b>Lessons Learnt</b>	<b>26</b>
7.1	Abhishek . . . . .	26
7.2	Anuraag . . . . .	26
7.3	Shardendu . . . . .	27
<b>8</b>	<b>Appendix</b>	<b>27</b>
8.1	scanner.mll . . . . .	27
8.2	parser.mly . . . . .	28
8.3	ast.ml . . . . .	33
8.4	semant.ml . . . . .	38
8.5	library.matcv . . . . .	61
8.6	codegen.ml . . . . .	63
8.7	matcv.ml . . . . .	84
8.8	Makefile . . . . .	84
<b>9</b>	<b>References</b>	<b>86</b>

# 1. Introduction

## 1.1 Motivation

MatCV is a programming language with type inference that aims at providing the programmers with a syntax that makes matrix manipulation easier and more intuitive. Since many fields, such as computer vision and machine learning use matrix operations extensively, our language introduces some constructs that allow beginners to get started easily. Instead of worrying about functions, scoping and types, users can straight away dive into working with matrices.

## 1.2 Description

MatCV is a language that makes it easier to work with and manipulate matrices. While other languages like Java have constructs to create multi-dimensional arrays, manipulating these arrays even for basic tasks is not as straightforward. Our language attempts to bridge this gap by making basic manipulation of n-dimensional matrices a fundamental part of our language. Our language also infers the types of the variables, so that the programmer can work without worrying about declaring variables.

## 1.3 Features

- Imperative Language
- Type Inference
- In-built Matrix Datatype
- Matrix-centric arithmetic operations
- Global variables

# 2. Tutorial

## 2.1 Running the compiler

Assuming OCaml, LLVM and Opam are installed, compiling can be done in the following steps:

1. Unpack the MatCV tar
2. Navigate to directory and make
3. To get the LLVM IR of a MatCV program called foo, execute the command

```
./matcv.native < foo
```

This will print the LLVM IR of foo. You can also use -l and -a options to generate the LLVM IR and generate the corresponding AST respectively.

## 2.2 Matrix Hello World

Hello World in our language is straightforward. The print statement can be invoked in either the global scope or inside a function. A sample program showing hello world in the global scope:

```
1 print(1);
```

Basic matrix operations can be performed in the global scope as well. Because the compiler infers types, there is no need to specify a type when declaring a variable, similar to Python. A sample is shown below:

```
1 i = 2;
2 a = {3,5;2,6};
3 b = {2,1;3,2};
4 c = a +. b;
```

Since our language focuses on matrices, here is a basic function definition that adds two matrices:

```
1 function addElements(matrix1, matrix2){
2
3     sum = matrix1 +. matrix2;
4
5     return sum;
6 }
```

## 3. Language Reference Manual

### 3.1 Data Types

MatCV supports the following data types:

int	64 bit integers (32 bit integers will not be supported)
boolean	true or false
matrix	m-by-n matrix which stores int/float type data
void	Data type used by functions that don't return anything

In MatCV, we do not need to explicitly specify the data type of the variable being declared. In case of invalid datatype conflict, MatCV throws an error.

```

1 a = 3; /* Type of a inferred as Int */
2 b = [3][4]; /* Type of b inferred as Mat(2)*/
3 /*which means that b is a matrix of 2 dimensions */
4 a + b; /* Throws mismatched type error: Type mismatch: Int, Mat(2)*/

```

### 3.2 Operators

While considering operations between data types, we enforce some restrictions on the data types that can be used with each other. The operators for are listed below:

Addition	+	Adds two expressions that can be int/boolean.
Subtraction	-	Subtracts two expressions that can be int/boolean
Multiplication	*	Multiplies two expressions that can be int
Remainder	%	Remainder obtained upon integer division.
Division	/	Divides an int/float
Assignment	=	We assign an appropriate RHS to an appropriate LHS
Equality Check	==	Returns 1 if two expressions are equal
Not Equal To	!=	Returns 1 if two expressions are not equal
Greater Than Operator	>	Compares the value of two expressions for
Greater Than or Equal To Operator	>=	Compares the value of two expressions
Less Than Operator	<	Compares the value of two expressions
Less Than Operator or Equal To	<=	Compares the value of two expressions
AND	&&	Logical AND Operator
OR		Logical OR Operator
MATPLUS	+.	Adds two matrices with same dimensions
MATMINUS	-.	Subtracts two matrices with same dimensions

To perform these operations on any two expressions, we can simply write them as:

```

1 a = 2;
2 b = 3;
3 a + b;

```

```
4 a*b;
5 a-b;
```

### 3.2.1 Matrix Operations

In case we want to perform addition, subtraction, multiplication or division between matrices, we need to use the MatPlus and the MatSub operators which are the normal operators followed by a period. The result of the addition and subtraction of two matrices is stored in the first operand. We have provided the user two ways to access matrices. If the user uses [ ] then they can access the array normally. They use the < > operator, to access the actual locations in which dimensions and sizes will be stored.

```
1 a = {1,2;3,4};
2 b = {5,6;7,8};
3 a +. b;
4 a -. b;
```

### 3.2.2 Operator Precedence

The precedence of our operators is the following from **Highest** to **Lowest** :

{ }, [ ]	Highest
!	
*, /, %	
+, -	
< , > , <= , >=	
== , !=	
& &	
=	Lowest

### 3.3 Comments

Multi - line and nested comments are supported:

```
/* This is a comment. Comments can be nested
and can be spread across multiple lines.
Comments have to be closed */
```

### 3.4 Keywords

MatCV supports the following keywords:

if..else if..else	Supports standard conditional operations
for	loops over given elements
break	breaks out of loop
continue	returns control flow to the beginning of the loop
exit	stops the program execution and returns control to the host environment
return	finish function execution and return value to the calling function

Their usage is as follows:

```

1  if (d == 3){
2      d = d + 1;
3  }
4  else{
5      d = d - 1;
6  } /* else is optional in the conditional check */
7
8  if (d == 3)
9      d = d + 1;
10 /* braces are also optional*/
11
12 for (i = 1; i < 10; i = i+1 ){
13     d = d + 1;
14 } /* For loop - all expressions are optional in the loop initialization */
15
16
17 for (; d < 10;){
18     d = d + 1;
19     if (d == 6){
20         continue; /* goes to beginning of of the loop */
21     }
22 }
23
24 function test(){
25     a = 2;
26     return a; /* returns control back to the calling function */
27 }

```

### 3.5 Identifiers

Identifiers in MatCV are alphanumeric and must must start with an alphabet.



## 3.6 Library Functions

Library functions are written in our language. These library functions are used to perform operations that are crucial to matrices - copying a matrix, addition and subtraction. These features can be used as follows:

```
1 a = {1,3;2,4};
2 b = {1,1;0,0};
3 a +. b; /* stores a + b in a */
4 a -. b; /* stores a - b in a */
5 a = b; /* copies contents of a to b */
```

## 3.7 Matrix Initialization

Matrices can be initialized in the following ways:

```
a = [3][3]; /* Creates a matrix of 3 x 3 dimensions */
a = {1,2;3,4}; /* Creates a matrix of 2 x 2 dimensions */
a = {1,2;3,4};
b = a; /* Initialized b with the dimension and values of a */
```

## 3.8 Functions

Functions have to be declared with the function keyword followed by the function name. Functions do not necessarily require a return statement. Code can be outside any of the functions as well.

```
function test(arg){
    a = arg*arg;
    return a;
}
```

## 3.9 Structure

1. All statements in MatCV are terminated by a semi-colon (;).
2. There is no specific function like 'main' that serves as the entry point in the program. Execution begins from the first statement in the program.
3. Blocks of code used by functions, if-else, for loops etc. have to be enclosed within opening and closing braces i.e. { and }
4. Variables declared outside the scope of any function belong to the global scope

## 4. Project Plan

### 4.1 Process

The team met weekly to discuss tasks and progress. We had a meetly meeting with our TA on Tuesdays at 5:30 PM, which helped us to be on track for the project and cleared whatever questions we had. The weekly team meetings helped us gauge progress of other team members and resolved issues that any team member might be facing. Sometimes the team faced 'bottlenecks', i.e. waiting for a module that was under development. This sometimes impeded progress but was solved with mostly discussion and helping out with difficult modules.

### 4.2 Team Roles

- Manager - Anuraag Advani
  - Responsible for scheduling meetings and tracking progress
  - Worked on parts of the scanner and parser
  - Worked on the code generator
- Language Guru - Abhishek Walia
  - Guided the broad specifications of the language
  - Built the type inference and semantic checker
  - Worked on the code generator
- System Architect - Shardendu Gautam
  - Responsible for development of architecture
  - Worked on the code generator and AST printing
  - Designed the test suite

Towards the end of the project, the roles became fluid as team members collaborated to work on crucial modules and deliverables.

### 4.3 Programming Style Guide

We tried following the important Ocaml style conventions:

- Keep the code readable and understandable. Since other members would be using and interfacing with your code, readability is important
- Follow camelCase for naming variables
- Keep variable names meaningful
- Keep consistent indentation and spacing across the project

## 4.4 Project Timeline

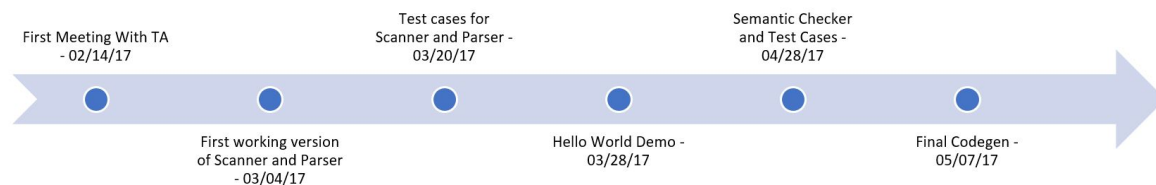


Figure 1: Timeline of project

## 4.5 Challenges

The primary challenges we faced in the project was the process of getting familiar with OCaml. The functional paradigm was new for all of us and it took time to get used to and achieve a certain level of proficiency that made us confident in the language.

It also took us a lot of time to get matrices to work to satisfaction. Implementing multi-dimensional matrices was particularly challenging and needed a deep understanding of LLVM and how it allows to store and retrieve data.

Since it is a semester long project, it was also challenging to gauge progress and stay on track. It was easy to lose track and fall behind, which did happen a few times, and needed extra effort from all members.

## 4.6 Development Environment

- Language - **Ocaml Version 4.01.0**  
**LLVM 3.6.8**
- Operating System - **Ubuntu subsystem for Windows**
- Editor - **Sublime Text/Vim**
- Version Control - **git**
- Test Scripts - **bash**

## 5. Architecture

The compiler for MatCV has the following components:

1. Scanner
2. Parser
3. Semantic Analyzer

4. MatCV Library

5. Code Generator

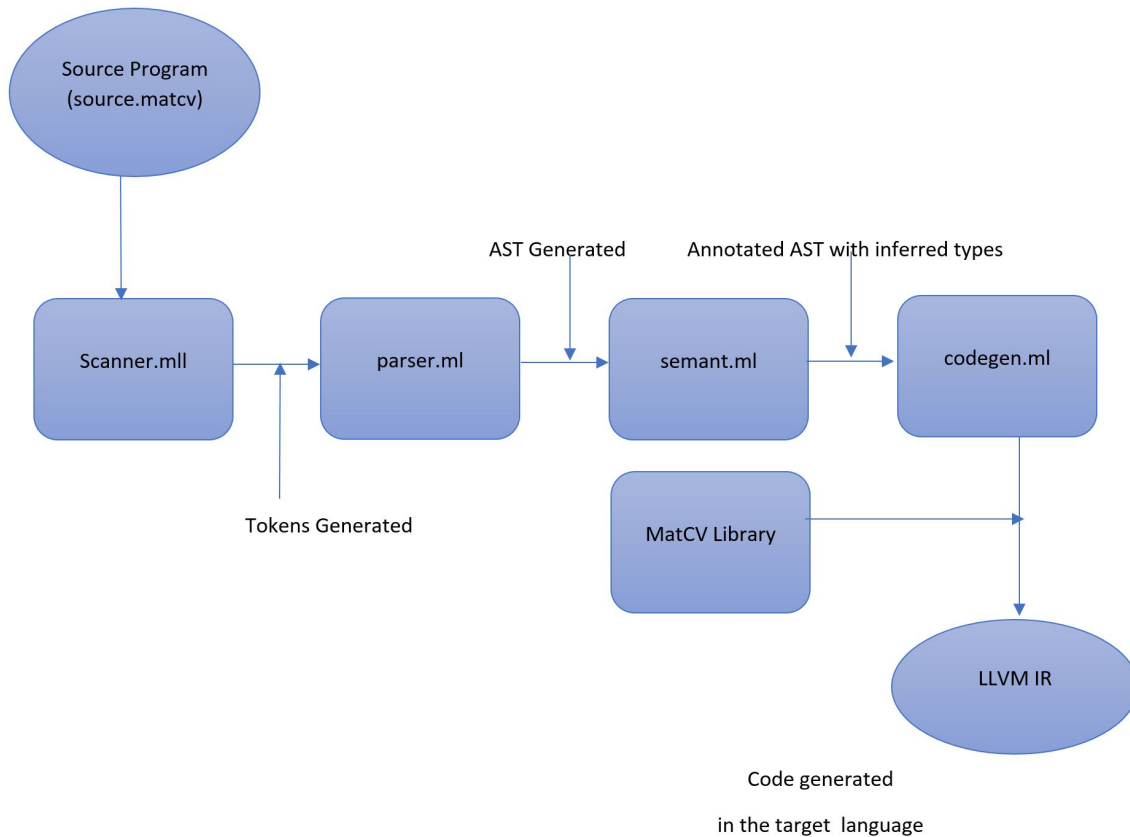


Figure 2: Architecture diagram for the compiler

The individual components of the compiler have the following tasks:

### 5.1 Scanner

The scanner is responsible for converting the code in the input file to tokens. These tokens are obtained by removing whitespaces, tabs and newlines. It establishes the proper nesting of the comments and discards them.

### 5.2 Parser

The parser takes the tokenized output from the scanner and checks it against the defined grammar. If the input program does not match any of the given grammar rules, it throws up a parsing error. If the parser does not spot any grammar violations, it constructs an Abstract Syntax Tree(AST).

### 5.3 Semantic Checker

The Abstract Syntax Tree is passed on to the semantic checker. For MatCV, the AST performs the following tasks:

- Since our code doesn't need users to define types, we need to infer types to ensure no type mismatch violations are taking place. The semantic checker assigns types to all variables and throws an error if the type of any expression can not be resolved.
- It also performs basic compile-type checks for matrices including getting the dimension of the matrix and raising error for incorrect dimensions
- It checks that no identifiers use the keywords defined in our language
- It also raises an error if statements are used incorrectly eg. continue outside a loop

```
a = {2,3,4;1,2};  
b = {2,3;4,2};
```

```
Error: Invalid dimensions were specified for Matrix: a
```

Figure 3: Invalid Dimensions throw a error [Error]

```
for (i = 0; i<3; i=i+1){  
    d = 0;  
}  
break;
```

```
Error: Invalid use of break.
```

Figure 4: Break cannot be outside a for loop [Error]

```
function foo(a,b){  
    c = a+b;  
    return c;  
}  
  
function foo(a){  
    b = 1;  
}
```

```
Error: Multiple definitions of function: foo
```

Figure 5: Semant Catches duplicate function names [Error]

The semantic checker passes the annotated AST which has the types of expressions to the codegen.

## 5.4 MatCV Library

We have written a library in MatCV to facilitate the manipulation of matrices. This library adds the functionality of addition, subtraction and copying a matrix to another matrix. This library is linked to codegen and is called when any of these functions is invoked. The library is written in MatCV and compiles to OCaml.

## 5.5 Code Generator

The code generator obtains the annotated AST from the semantic checker. Our codegen compiles to LLVM IR as the target language using the OCaml LLVM module. The codegen iterates through the annotated AST and converts the expressions and statements to LLVM code. It also throws up any run-time errors that could not have been checked for by the semantic checker. If there are no errors, the codegen outputs LLVM code.

# 6. Testing

## 6.1 Test Plan

After the completion of a checkpoint of each module, we developed test cases for those modules to ensure that any future changes to the module don't break existing functions. The tests were written in such a way to ensure that we have a mix of tests that are expected to pass and fail. As things were added and removed from the modules over the course of the project, the test cases had to evolve as well. We wrote a lot of small code snippets that tested the individual features of the language. This was to ensure that any errors could be identified easily.

## 6.2 Automation

We relied on testing our changes to the code by running scripts that ran our test suite for all the modules. We use a script to run the tests for all modules. Instead of using Travis CI as a testing framework, we relied on executing the scripts manually before pushing any commits to the repository.

We use the following script:

```
1 #!/bin/bash
2
3 TEST_FILES=`find ./Tests -type f -not -name "*.out"`
4
5
```

```

6 echo "Test $I)"
7 if [ ! -f matcv.native ]
8 then
9     make
10 fi
11
12 I=0
13 for FILE in $TEST_FILES
14 do
15     echo "$I) Running test case for $FILE:"
16
17     cat library.matcv $FILE > __run__
18     ./matcv.native < __run__ &> __out__
19     if [ $? != 0 ]
20     then
21         echo "Test failed for1: $FILE"
22     else
23         lli __out__ &>> __output__
24         if [ $? != 0 ]
25         then
26             echo "Test failed for2: $FILE"
27         else
28             diff __output__ "$FILE".out
29             if [ $? != 0 ]
30             then
31                 echo "Test failed for2: $FILE"
32             else
33                 echo "Test passed for: $FILE"
34             fi
35         fi
36     fi
37
38     I=`expr $I + 1`
39
40
41     rm -f __run__ __out__ __output__
42
43     echo "-----"
44
45
46 done

```

## 6.3 Demo Code - Source Code to Target Language

### 6.4 Program 1

This program takes a one-dimensional matrix of the different denomination of coins we have and a target sum. Given these arguments, it calculates the number of ways we can obtain the target sum.

#### Source Code

```
1 A = [3];
2 a[0] = 1;
3 a[1] = 2;
4 a[2] = 3;
5
6 n = 4;
7 m = 3;
8
9 result = coinChange(a,m,n);
10 print(result);
11
12 function coinChange(S, m, n){
13     table = [n+1][m];
14     x = 0;
15     i = 0;
16     j = 0;
17     y = 0;
18     for (i = 0; i < m; i=i+1)
19     {
20         table[0][i] = 1;
21     }
22
23     for (i = 1; i < n+1; i=i+1)
24     {
25         for (j = 0; j < m; j=j+1)
26         {
27             if (i - S[j] >= 0){
28                 temp = i - S[j];
29                 x = table[temp][j];
30             }
31             else{
32                 x = 0;
33             }
34             if (j >= 1){
35
36                 y = table[i][j-1];
```



```

37         }
38         else{
39             y = 0;
40         }
41         table[i][j] = x + y;
42     }
43 }
44 ans = table[n][m-1];
45 return ans;

```

## Target Language - LLVM IR

```

1 ; ModuleID = 'MatCV'
2
3 @a = global i32* null
4 @n = global i32 0
5 @m = global i32 0
6 @result = global i32 0
7 @fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
8 @fmt1 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
9
10 define i32 @main() {
11     entry:
12         %a_free = load i32** @a
13         %0 = bitcast i32* %a_free to i8*
14         tail call void @free(i8* %0)
15         %mallocall = tail call i8* @malloc(i32 mul (i32 ptrtoint (i32*
16             ↪ getelementptr (i32* null, i32 1) to i32), i32 5))
17         %a_malloc = bitcast i8* %mallocall to i32*
18         store i32* %a_malloc, i32** @a
19         %a_load = load i32** @a
20         %a_zero_index = getelementptr inbounds i32* %a_load, i32 0
21         store i32 5, i32* %a_zero_index
22         %a_ptr_index = getelementptr inbounds i32* %a_load, i32 1
23         store i32 3, i32* %a_ptr_index
24         %a_load1 = load i32** @a
25         %a_dim_1 = getelementptr inbounds i32* %a_load1, i32 1
26         %a_dim_1value_ = load i32* %a_dim_1
27         %tmp3_ = mul i32 1, %a_dim_1value_
28         %a_element = getelementptr inbounds i32* %a_load1, i32 2
29         store i32 1, i32* %a_element
30         %a_load2 = load i32** @a
31         %a_dim_13 = getelementptr inbounds i32* %a_load2, i32 1
32         %a_dim_1value_4 = load i32* %a_dim_13

```

```

32 %tmp3_5 = mul i32 1, %a_dim_1value_4
33 %a_element6 = getelementptr inbounds i32* %a_load2, i32 3
34 store i32 2, i32* %a_element6
35 %a_load7 = load i32** @a
36 %a_dim_18 = getelementptr inbounds i32* %a_load7, i32 1
37 %a_dim_1value_9 = load i32* %a_dim_18
38 %tmp3_10 = mul i32 1, %a_dim_1value_9
39 %a_element11 = getelementptr inbounds i32* %a_load7, i32 4
40 store i32 3, i32* %a_element11
41 store i32 4, i32* @n
42 store i32 3, i32* @m
43 %a = load i32** @a
44 %m = load i32* @m
45 %n = load i32* @n
46 %coinChange_result = call i32 @coinChange(i32* %a, i32 %m, i32 %n)
47 store i32 %coinChange_result, i32* @result
48 %result = load i32* @result
49 %printf = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4
  ↪ x i8]* @fmt, i32 0, i32 0), i32 %result)
50 ret i32 0
51 }
52
53 define i32 @coinChange(i32* %S, i32 %m, i32 %n) {
54 entry:
55 %S1 = alloca i32*
56 store i32* %S, i32** %S1
57 %m2 = alloca i32
58 store i32 %m, i32* %m2
59 %n3 = alloca i32
60 store i32 %n, i32* %n3
61 %n4 = load i32* %n3
62 %tmp = add i32 %n4, 1
63 %expr_prod_size = mul i32 1, %tmp
64 %m5 = load i32* %m2
65 %expr_prod_size6 = mul i32 %expr_prod_size, %m5
66 %mat_size = add i32 %expr_prod_size6, 3
67 %table = alloca i32*
68 %malloccsize = mul i32 %mat_size, ptrtoint (i32* getelementptr (i32*
  ↪ null, i32 1) to i32)
69 %malloccall = tail call i8* @malloc(i32 %malloccsize)
70 %table_malloc = bitcast i8* %malloccall to i32*
71 store i32* %table_malloc, i32** %table
72 %table_load = load i32** %table

```

```

73  %table_zero_index = getelementptr inbounds i32* %table_load, i32 0
74  store i32 %mat_size, i32* %table_zero_index
75  %table_ptr_index = getelementptr inbounds i32* %table_load, i32 1
76  store i32 %tmp, i32* %table_ptr_index
77  %table_ptr_index7 = getelementptr inbounds i32* %table_load, i32 2
78  store i32 %m5, i32* %table_ptr_index7
79  %x = alloca i32
80  store i32 0, i32* %x
81  %i = alloca i32
82  store i32 0, i32* %i
83  %j = alloca i32
84  store i32 0, i32* %j
85  %y = alloca i32
86  store i32 0, i32* %y
87  store i32 0, i32* %i
88  br label %for
89
90  forincr:                                     ; preds = %forbody
91  %i8 = load i32* %i
92  %tmp9 = add i32 %i8, 1
93  store i32 %tmp9, i32* %i
94  br label %for
95
96  for:                                         ; preds = %forincr,
    ↪ %entry
97  %i10 = load i32* %i
98  %m11 = load i32* %m2
99  %tmp12 = icmp slt i32 %i10, %m11
100 br i1 %tmp12, label %forbody, label %merge
101
102 forbody:                                     ; preds = %for
103 %i13 = load i32* %i
104 %table_load14 = load i32** %table
105 %table_dim_2 = getelementptr inbounds i32* %table_load14, i32 2
106 %table_dim_2value_ = load i32* %table_dim_2
107 %table_dim_1 = getelementptr inbounds i32* %table_load14, i32 1
108 %table_dim_1value_ = load i32* %table_dim_1
109 %tmp_ = mul i32 1, %i13
110 %tmp2_ = add i32 3, %tmp_
111 %tmp3_ = mul i32 1, %table_dim_2value_
112 %tmp_15 = mul i32 %tmp3_, 0
113 %tmp2_16 = add i32 %tmp2_, %tmp_15
114 %tmp3_17 = mul i32 %tmp3_, %table_dim_1value_

```

```

115     %table_element = getelementptr @inbounds, i32* %table_load14, i32
      ↪ %tmp2_16
116     store i32 1, i32* %table_element
117     br label %forincr
118
119     merge:                                     ; preds = %for
120     store i32 1, i32* %i
121     br label %for21
122
123     forincr18:                                 ; preds = %merge108
124     %i19 = load i32* %i
125     %tmp20 = add i32 %i19, 1
126     store i32 %tmp20, i32* %i
127     br label %for21
128
129     for21:                                     ; preds = %forincr18,
      ↪ %merge
130     %i22 = load i32* %i
131     %n23 = load i32* %n3
132     %tmp24 = add i32 %n23, 1
133     %tmp25 = icmp slt i32 %i22, %tmp24
134     br i1 %tmp25, label %forbody26, label %merge109
135
136     forbody26:                                 ; preds = %for21
137     store i32 0, i32* %j
138     br label %for30
139
140     forincr27:                                 ; preds = %merge72
141     %j28 = load i32* %j
142     %tmp29 = add i32 %j28, 1
143     store i32 %tmp29, i32* %j
144     br label %for30
145
146     for30:                                     ; preds = %forincr27,
      ↪ %forbody26
147     %j31 = load i32* %j
148     %m32 = load i32* %m2
149     %tmp33 = icmp slt i32 %j31, %m32
150     br i1 %tmp33, label %forbody34, label %merge108
151
152     forbody34:                                 ; preds = %for30
153     %i35 = load i32* %i
154     %j36 = load i32* %j
155     %S_load = load i32** %S1

```

```

156 %S_dim_1 = getelementptr inbounds i32* %S_load, i32 1
157 %S_dim_1value_ = load i32* %S_dim_1
158 %tmp_37 = mul i32 1, %j36
159 %tmp2_38 = add i32 2, %tmp_37
160 %tmp3_39 = mul i32 1, %S_dim_1value_
161 %S_element = getelementptr inbounds i32* %S_load, i32 %tmp2_38
162 %S_element40 = load i32* %S_element
163 %tmp41 = sub i32 %i35, %S_element40
164 %tmp42 = icmp sge i32 %tmp41, 0
165 br i1 %tmp42, label %then, label %else
166
167 merge43: ; preds = %else,
    ↪ %then
168 %j70 = load i32* %j
169 %tmp71 = icmp sge i32 %j70, 1
170 br i1 %tmp71, label %then73, label %else90
171
172 then: ; preds = %forbody34
173 %temp = alloca i32
174 %i44 = load i32* %i
175 %j45 = load i32* %j
176 %S_load46 = load i32** %S1
177 %S_dim_147 = getelementptr inbounds i32* %S_load46, i32 1
178 %S_dim_1value_48 = load i32* %S_dim_147
179 %tmp_49 = mul i32 1, %j45
180 %tmp2_50 = add i32 2, %tmp_49
181 %tmp3_51 = mul i32 1, %S_dim_1value_48
182 %S_element52 = getelementptr inbounds i32* %S_load46, i32 %tmp2_50
183 %S_element53 = load i32* %S_element52
184 %tmp54 = sub i32 %i44, %S_element53
185 store i32 %tmp54, i32* %temp
186 %temp55 = load i32* %temp
187 %j56 = load i32* %j
188 %table_load57 = load i32** %table
189 %table_dim_258 = getelementptr inbounds i32* %table_load57, i32 2
190 %table_dim_2value_59 = load i32* %table_dim_258
191 %table_dim_160 = getelementptr inbounds i32* %table_load57, i32 1
192 %table_dim_1value_61 = load i32* %table_dim_160
193 %tmp_62 = mul i32 1, %j56
194 %tmp2_63 = add i32 3, %tmp_62
195 %tmp3_64 = mul i32 1, %table_dim_2value_59
196 %tmp_65 = mul i32 %tmp3_64, %temp55
197 %tmp2_66 = add i32 %tmp2_63, %tmp_65
198 %tmp3_67 = mul i32 %tmp3_64, %table_dim_1value_61

```

```

199  %table_element68 = getelementptr inbounds i32* %table_load57, i32
    ↪ %tmp2_66
200  %table_element69 = load i32* %table_element68
201  store i32 %table_element69, i32* %x
202  br label %merge43
203
204  else:                                     ; preds = %forbody34
205  store i32 0, i32* %x
206  br label %merge43
207
208  merge72:                                   ; preds = %else90,
    ↪ %then73
209  %i91 = load i32* %i
210  %j92 = load i32* %j
211  %table_load93 = load i32** %table
212  %table_dim_294 = getelementptr inbounds i32* %table_load93, i32 2
213  %table_dim_2value_95 = load i32* %table_dim_294
214  %table_dim_196 = getelementptr inbounds i32* %table_load93, i32 1
215  %table_dim_1value_97 = load i32* %table_dim_196
216  %tmp_98 = mul i32 1, %j92
217  %tmp2_99 = add i32 3, %tmp_98
218  %tmp3_100 = mul i32 1, %table_dim_2value_95
219  %tmp_101 = mul i32 %tmp3_100, %i91
220  %tmp2_102 = add i32 %tmp2_99, %tmp_101
221  %tmp3_103 = mul i32 %tmp3_100, %table_dim_1value_97
222  %table_element104 = getelementptr inbounds i32* %table_load93, i32
    ↪ %tmp2_102
223  %x105 = load i32* %x
224  %y106 = load i32* %y
225  %tmp107 = add i32 %x105, %y106
226  store i32 %tmp107, i32* %table_element104
227  br label %forincr27
228
229  then73:                                     ; preds = %merge43
230  %i74 = load i32* %i
231  %j75 = load i32* %j
232  %tmp76 = sub i32 %j75, 1
233  %table_load77 = load i32** %table
234  %table_dim_278 = getelementptr inbounds i32* %table_load77, i32 2
235  %table_dim_2value_79 = load i32* %table_dim_278
236  %table_dim_180 = getelementptr inbounds i32* %table_load77, i32 1
237  %table_dim_1value_81 = load i32* %table_dim_180
238  %tmp_82 = mul i32 1, %tmp76
239  %tmp2_83 = add i32 3, %tmp_82

```

```

240   %tmp3_84 = mul i32 1, %table_dim_2value_79
241   %tmp_85 = mul i32 %tmp3_84, %i74
242   %tmp2_86 = add i32 %tmp2_83, %tmp_85
243   %tmp3_87 = mul i32 %tmp3_84, %table_dim_1value_81
244   %table_element88 = getelementptr inbounds i32* %table_load77, i32
    ↪ %tmp2_86
245   %table_element89 = load i32* %table_element88
246   store i32 %table_element89, i32* %y
247   br label %merge72
248
249   else90:                                     ; preds = %merge43
250   store i32 0, i32* %y
251   br label %merge72
252
253   merge108:                                   ; preds = %for30
254   br label %forincr18
255
256   merge109:                                   ; preds = %for21
257   %ans = alloca i32
258   %n110 = load i32* %n3
259   %m111 = load i32* %m2
260   %tmp112 = sub i32 %m111, 1
261   %table_load113 = load i32** %table
262   %table_dim_2114 = getelementptr inbounds i32* %table_load113, i32 2
263   %table_dim_2value_115 = load i32* %table_dim_2114
264   %table_dim_1116 = getelementptr inbounds i32* %table_load113, i32 1
265   %table_dim_1value_117 = load i32* %table_dim_1116
266   %tmp_118 = mul i32 1, %tmp112
267   %tmp2_119 = add i32 3, %tmp_118
268   %tmp3_120 = mul i32 1, %table_dim_2value_115
269   %tmp_121 = mul i32 %tmp3_120, %n110
270   %tmp2_122 = add i32 %tmp2_119, %tmp_121
271   %tmp3_123 = mul i32 %tmp3_120, %table_dim_1value_117
272   %table_element124 = getelementptr inbounds i32* %table_load113, i32
    ↪ %tmp2_122
273   %table_element125 = load i32* %table_element124
274   store i32 %table_element125, i32* %ans
275   %ans126 = load i32* %ans
276   ret i32 %ans126
277 }
278
279 declare i32 @printf(i8*, ...)
280
281 declare i32 @copyMat(i32*, i32*)

```

```

282
283 declare i32 @minusMat(i32*, i32*)
284
285 declare i32 @addMat(i32*, i32*)
286
287 declare void @free(i8*)
288
289 declare noalias i8* @malloc(i32)

```

## 6.5 Program 2

This code snippet illustrates the core functionality of our language. It shows simple matrix addition and subtraction in which the result is stored in the first operand. We then print the first element of the result.

### Source Code

```

1  a = {1,2,3;4,5,6;7,8,9};
2  b = {9,8,7;6,5,4;3,2,1};
3
4  a +. b;
5  printMatrix(a);
6  a -. b;
7  printMatrix(a);
8
9
10 function printMatrix(mat)
11 {
12     print (mat[0][0]);
13     return 0;
14 }

```

### Target Language - LLVM IR

```

1 ; ModuleID = 'MatCV'
2
3 @a = global i32* null
4 @b = global i32* null
5 @fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
6 @fmt1 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
7
8 define i32 @main() {
9     entry:
10     %a_free = load i32** @a
11     %0 = bitcast i32* %a_free to i8*

```



```

12 tail call void @free(i8* %0)
13 %malloccall = tail call i8* @malloc(i32 mul (i32 ptrtoint (i32*
   ↪ getelementptr (i32* null, i32 1) to i32), i32 12))
14 %a_malloc = bitcast i8* %malloccall to i32*
15 store i32* %a_malloc, i32** @a
16 %a_load = load i32** @a
17 %a_zero_index = getelementptr inbounds i32* %a_load, i32 0
18 store i32 2, i32* %a_zero_index
19 %a_one_index = getelementptr inbounds i32* %a_load, i32 1
20 store i32 3, i32* %a_one_index
21 %a_two_index = getelementptr inbounds i32* %a_load, i32 2
22 store i32 3, i32* %a_two_index
23 %a_ptr_index = getelementptr inbounds i32* %a_load, i32 3
24 store i32 1, i32* %a_ptr_index
25 %a_ptr_index1 = getelementptr inbounds i32* %a_load, i32 4
26 store i32 2, i32* %a_ptr_index1
27 %a_ptr_index2 = getelementptr inbounds i32* %a_load, i32 5
28 store i32 3, i32* %a_ptr_index2
29 %a_ptr_index3 = getelementptr inbounds i32* %a_load, i32 6
30 store i32 4, i32* %a_ptr_index3
31 %a_ptr_index4 = getelementptr inbounds i32* %a_load, i32 7
32 store i32 5, i32* %a_ptr_index4
33 %a_ptr_index5 = getelementptr inbounds i32* %a_load, i32 8
34 store i32 6, i32* %a_ptr_index5
35 %a_ptr_index6 = getelementptr inbounds i32* %a_load, i32 9
36 store i32 7, i32* %a_ptr_index6
37 %a_ptr_index7 = getelementptr inbounds i32* %a_load, i32 10
38 store i32 8, i32* %a_ptr_index7
39 %a_ptr_index8 = getelementptr inbounds i32* %a_load, i32 11
40 store i32 9, i32* %a_ptr_index8
41 %b_free = load i32** @b
42 %1 = bitcast i32* %b_free to i8*
43 tail call void @free(i8* %1)
44 %malloccall9 = tail call i8* @malloc(i32 mul (i32 ptrtoint (i32*
   ↪ getelementptr (i32* null, i32 1) to i32), i32 12))
45 %b_malloc = bitcast i8* %malloccall9 to i32*
46 store i32* %b_malloc, i32** @b
47 %b_load = load i32** @b
48 %b_zero_index = getelementptr inbounds i32* %b_load, i32 0
49 store i32 2, i32* %b_zero_index
50 %b_one_index = getelementptr inbounds i32* %b_load, i32 1
51 store i32 3, i32* %b_one_index
52 %b_two_index = getelementptr inbounds i32* %b_load, i32 2

```

```

53 store i32 3, i32* %b_two_index
54 %b_ptr_index = getelementptr inbounds i32* %b_load, i32 3
55 store i32 9, i32* %b_ptr_index
56 %b_ptr_index10 = getelementptr inbounds i32* %b_load, i32 4
57 store i32 8, i32* %b_ptr_index10
58 %b_ptr_index11 = getelementptr inbounds i32* %b_load, i32 5
59 store i32 7, i32* %b_ptr_index11
60 %b_ptr_index12 = getelementptr inbounds i32* %b_load, i32 6
61 store i32 6, i32* %b_ptr_index12
62 %b_ptr_index13 = getelementptr inbounds i32* %b_load, i32 7
63 store i32 5, i32* %b_ptr_index13
64 %b_ptr_index14 = getelementptr inbounds i32* %b_load, i32 8
65 store i32 4, i32* %b_ptr_index14
66 %b_ptr_index15 = getelementptr inbounds i32* %b_load, i32 9
67 store i32 3, i32* %b_ptr_index15
68 %b_ptr_index16 = getelementptr inbounds i32* %b_load, i32 10
69 store i32 2, i32* %b_ptr_index16
70 %b_ptr_index17 = getelementptr inbounds i32* %b_load, i32 11
71 store i32 1, i32* %b_ptr_index17
72 %a = load i32** @a
73 %b = load i32** @b
74 %addMat = call i32 @addMat(i32* %a, i32* %b)
75 %a18 = load i32** @a
76 %printMatrix_result = call i32 @printMatrix(i32* %a18)
77 %a19 = load i32** @a
78 %b20 = load i32** @b
79 %minusMat = call i32 @minusMat(i32* %a19, i32* %b20)
80 %a21 = load i32** @a
81 %printMatrix_result22 = call i32 @printMatrix(i32* %a21)
82 ret i32 0
83 }
84
85 define i32 @printMatrix(i32* %mat) {
86 entry:
87 %mat1 = alloca i32*
88 store i32* %mat, i32** %mat1
89 %mat_load = load i32** %mat1
90 %mat_dim_2 = getelementptr inbounds i32* %mat_load, i32 2
91 %mat_dim_2value_ = load i32* %mat_dim_2
92 %mat_dim_1 = getelementptr inbounds i32* %mat_load, i32 1
93 %mat_dim_1value_ = load i32* %mat_dim_1
94 %tmp3_ = mul i32 1, %mat_dim_2value_
95 %tmp_ = mul i32 %tmp3_, 0

```

```

96  %tmp2_ = add i32 3, %tmp_
97  %tmp3_2 = mul i32 %tmp3_, %mat_dim_1value_
98  %mat_element = getelementptr inbounds i32* %mat_load, i32 %tmp2_
99  %mat_element3 = load i32* %mat_element
100 %printf = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4
    ↪ x i8]* @fmt1, i32 0, i32 0), i32 %mat_element3)
101 ret i32 0
102 }
103
104 declare i32 @printf(i8*, ...)
105
106 declare i32 @copyMat(i32*, i32*)
107
108 declare i32 @minusMat(i32*, i32*)
109
110 declare i32 @addMat(i32*, i32*)
111
112 declare void @free(i8*)
113
114 declare noalias i8* @malloc(i32)

```

## 7. Lessons Learnt

### 7.1 Abhishek

Designing and implementing a compiler is a challenging but fulfilling task and OCaml makes everything so much easier. Learning how to perform type inference and generate LLVM IR code was a great experience. This is the first time I have coded in a functional language and I thoroughly enjoyed it. Writing a compiler has made me realize that language design is the key. It is easy to come up with something that is cool, but it is excruciatingly difficult to come up with something that will stick for a long time.

### 7.2 Anuraag

This was one of the most challenging projects I have worked on. But I think it was worth all the effort as we ended up creating our own programming language. Though we did not implement everything we planned I am really happy with the features in our language. I feel the features that we implemented gave us a really good idea about how to build a good compiler.

## 7.3 Shardendu

Working with a completely new programming paradigm seemed daunting at first but during the course of the project, I got accustomed to OCaml. The key was to write a lot of code and learn by doing. Overall, the experience of building a programming language was supremely educating. Tip to future teams: Choose your project scope carefully, get comfortable with OCaml in the beginning, don't underestimate the power of things messing up when you least expect them to.

## 8. Appendix

### 8.1 scanner.mll

```
1  (* MatCV scanner *)
2
3  { open Parser }
4
5  rule token = parse
6  | [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
7  | "/*"      { comment 0 lexbuf }      (* Comments *)
8  | '('      { LPAREN }
9  | ')'     { RPAREN }
10 | '{'     { LBRACE }
11 | '}'     { RBRACE }
12 | '['     { LSQBRACKET }
13 | ']'     { RSQBRACKET }
14 | ':'     { COLON }
15 | ';'     { SEMI }
16 | ','     { COMMA }
17 | '.'     { DOT }
18 | '+'     { PLUS }
19 | '-'     { MINUS }
20 | '*'     { TIMES }
21 | '/'     { DIVIDE }
22 | "+."    { MATPLUS }
23 | "-."    { MATMINUS }
24 | "*."    { MATTIMES }
25 | "/."    { MATDIVIDE }
26 | '%'     { MOD }
27 | '^'     { EXP }
28 | '='     { ASSIGN }
29 | "=="    { EQ }
30 | "!="    { NEQ }
31 | '<'     { LT }
```

```

32 | "<="      { LEQ }
33 | ">"       { GT }
34 | ">="      { GEQ }
35 | "&&"      { AND }
36 | "||"     { OR }
37 | "!"      { NOT }
38 | "row"    { ROW }
39 | "col"    { COL }
40 | "ele"    { ELE }
41 | "pixel"  { PIXEL }
42 | "var"    { VARKEYWORD }
43 | "const"  { CONSTANT }
44 | "if"     { IF }
45 | "else"   { ELSE }
46 | "for"    { FOR }
47 | "break"  { BREAK }
48 | "continue" { CONTINUE }
49 | "exit"   { EXIT }
50 | "while"  { WHILE }
51 | "return" { RETURN }
52 | "function" { FUNCTION }
53 | "true"   { TRUE }
54 | "false"  { FALSE }
55 | ['0'-'9']+ as lexeme { LITERAL(int_of_string lexeme) }
56 | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lexeme {
   | ↪ ID(lexeme) }
57 | eof { EOF }
58 | _ as char { raise (Failure("Illegal Character: " ^ Char.escaped
   | ↪ char)) }
59
60 (For nested comments *)
61 and comment level = parse
62 | "/*" {comment (level + 1) lexbuf}
63 | "*/" { if level = 0 then token lexbuf else comment (level - 1)
   | ↪ lexbuf}
64 | _ { comment level lexbuf }

```

## 8.2 parser.mly

```

1 /* MatCV Parser */
2 %{ open Ast %}
3
4 /***** TODO *****/
5 /*****MATRIX SPLICING? *****/

```

```

6
7
8
9
10 %token LPAREN RPAREN LBRACE RBRACE LSQBRACKET RSQBRACKET
11 %token COLON SEMI COMMA DOT PLUS MINUS TIMES DIVIDE
12 %token MATPLUS MATMINUS MATTIMES MATDIVIDE
13 %token MOD EXP ASSIGN EQ NEQ LT LEQ GT GEQ AND OR
14 %token NOT ROW COL ELE PIXEL VARKEYWORD CONSTANT
15 %token IF ELSE FOR BREAK CONTINUE EXIT WHILE RETURN
16 %token FUNCTION TRUE FALSE EOF
17 %token <int> LITERAL
18 %token <string> ID
19
20 %left SEMI
21 %left COMMA
22 %nonassoc NOELSE
23 %nonassoc ELSE
24 %right ASSIGN
25 %left OR
26 %left AND
27 %left EQ NEQ
28 %left LT GT LEQ GEQ
29 %left PLUS MINUS MATPLUS MATMINUS
30 %left TIMES DIVIDE MOD MATTIMES MATDIVIDE
31 %right NOT NEG
32 %right EXP
33 %left DOT
34 %nonassoc UNBOUNDED
35
36
37
38 %start program
39 %type <Ast.program> program
40
41
42 %%
43
44 program:
45 statements EOF { List.rev (fst $1), snd $1 }
46
47 statements:
48 /*nothing*/ { [], [] }
49 |statements statement { ($2 :: fst $1), snd $1 }

```

```

50     |statements functionDefinition { fst $1, ($2 :: snd $1) }
51
52 statementList:
53 /*nothing*/ { [] }
54 | statementList statement { $2 :: $1 }
55
56
57 functionDefinition:
58 FUNCTION ID LPAREN formalArguments RPAREN LBRACE statementList RBRACE {
59     {fname = $2;
60         formals = List.rev $4;
61         body = List.rev $7}}
62
63 ifStatement:
64 IF LPAREN expr RPAREN statement %prec NOELSE { If($3, $5, Block([]))}
65 | IF LPAREN expr RPAREN statement ELSE statement { If($3, $5, $7) }
66
67 blockOfStatements:
68 LBRACE statementList RBRACE { Block(List.rev $2) }
69
70 forLoop:
71 FOR LPAREN optionalVarAssign SEMI optionalExpression SEMI
72   → optionalVarAssign RPAREN statement { For($3, $5, $7, $9) }
73
74 optionalVarAssign:
75     /*nothing*/ { Nodecl }
76 | variableDeclaration {$1}
77
78 optionalExpression:
79 /*nothing*/ { Noexpr }
80 | expr {$1}
81
82
83 whileLoop:
84 WHILE LPAREN expr RPAREN statement { While($3, $5) }
85
86
87 statement:
88     /*nothing*/
89 | blockOfStatements                { $1 }
90 | expr SEMI                        { Expr $1 }
91 | variableDeclaration SEMI         { VarDecl $1 }
92 | returnStatement SEMI             {$1}

```

```

93 | ifStatement                                {$1}
94 | forLoop                                    {$1}
95 | whileLoop                                  {$1}
96 | rowLoop                                    {$1}
97 | eleLoop                                    {$1}
98 | pixelLoop                                  {$1}
99 | EXIT SEMI                                  {Exit}
100 | BREAK SEMI                                 {Break}
101 | CONTINUE SEMI                              {Continue}
102
103
104 eleLoop:
105     ELE ID COLON ID statement { ForEachLoop ($2, $4, $5, Ele) }
106
107 rowLoop:
108     ROW ID COLON ID statement { ForEachLoop ($2, $4, $5, Row) }
109
110 pixelLoop:
111     PIXEL ID COLON ID statement { ForEachLoop ($2, $4, $5, Pixel) }
112
113
114 actualArguments:
115 /*nothing*/                                {[]}
116 | expr                                      { [$1] }
117 | actualArguments COMMA expr { $3 :: $1 }
118
119 formalArguments:
120 /*nothing*/ { [] }
121 | ID { [$1] }
122 | formalArguments COMMA ID { $3 :: $1 }
123
124 returnStatement:
125 RETURN expr { Return($2) }
126 | RETURN { Return(Noexpr) }
127
128
129 variableDeclaration:
130 ID ASSIGN LBRACE matrixInitValues RBRACE { Matrix($1, List.rev
    ↪ (List.rev (fst $4) :: snd $4)) }
131 | ID ASSIGN dimensions { DimAssign($1, List.rev $3) }
132 | ID ASSIGN expr { ExprAssign($1,$3) }
133 | ID dimensions ASSIGN expr { MatElementAssign($1, List.rev $2, $4) }
134
135

```



```

136 dimensions:
137     LSQBRACKET expr RSQBRACKET { [$2] }
138 | dimensions LSQBRACKET expr RSQBRACKET {$3 :: $1}
139
140
141 matrixInitValues:
142 | matrixInitValues COMMA expr { $3 :: fst $1, snd $1 }
143 | matrixInitValues SEMI expr { [$3], List.rev (fst $1) :: snd $1}
144 | expr { [$1], [] }
145
146 functionCall:
147 ID LPAREN actualArguments RPAREN {Call($1, List.rev $3)}
148
149 expr:
150     LITERAL { Literal($1) }
151 | TRUE      { BoolLit(true) }
152 | FALSE     { BoolLit(false) }
153 | ID        { Id($1) }
154 | ID MATPLUS ID      { MatPlus($1, $3) }
155 | ID MATMINUS ID     { MatMinus($1, $3) }
156 | LT ID COMMA expr GT %prec UNBOUNDED { UnboundedAccessRead($2, $4) }
157 | LSQBRACKET LSQBRACKET ID COMMA expr COMMA expr RSQBRACKET
   ↪ RSQBRACKET %prec UNBOUNDED{ UnboundedAccessWrite($3, $5, $7) }
158 | ID dimensions { MatAccess($1, List.rev $2) }
159 | expr PLUS expr { BinaryOp($1, Add, $3) }
160 | expr MINUS expr { BinaryOp($1, Sub, $3) }
161 | expr TIMES expr { BinaryOp($1, Mul, $3) }
162 | expr DIVIDE expr { BinaryOp($1, Div, $3) }
163 | expr MOD expr { BinaryOp($1, Mod, $3) }
164 | expr EQ expr { BinaryOp($1, Equal, $3) }
165 | expr NEQ expr { BinaryOp($1, Neq, $3) }
166 | expr LT expr { BinaryOp($1, Less, $3) }
167 | expr LEQ expr { BinaryOp($1, Leq, $3) }
168 | expr GT expr { BinaryOp($1, Greater, $3) }
169 | expr GEQ expr { BinaryOp($1, Geq, $3) }
170 | expr AND expr { BinaryOp($1, And, $3) }
171 | expr OR expr { BinaryOp($1, Or, $3) }
172 | expr EXP expr { BinaryOp($1, Exp, $3) }
173 | MINUS expr %prec NEG { Unop(Neg, $2) }
174 | NOT expr { Unop(Not, $2) }
175 | functionCall {$1}
176 | LPAREN expr RPAREN {$2}

```

## 8.3 ast.ml

```
1  (* MatCV AST *)
2
3  (* TODO *)
4
5  type op = Add | Sub | Mul | Div | Equal | Neq | Less | Leq | Greater | Geq | And | Or
6
7  type uop = Neg | Not
8
9  type loopType = Row | Ele | Pixel
10
11 type expr =
12     Literal of int
13   | BoolLit of bool
14   | Id of string
15   | UnboundedAccessRead of string * expr
16   | UnboundedAccessWrite of string * expr * expr
17   | MatPlus of string * string
18   | MatMinus of string * string
19   | MatAccess of string * expr list
20   | BinaryOp of expr * op * expr
21   | Unop of uop * expr
22   | Call of string * expr list
23   | Noexpr
24
25
26 type varDecl =
27     Nodecl
28   | Matrix of string * expr list list
29   | ExprAssign of string * expr
30   | DimAssign of string * expr list
31   | MatElementAssign of string * expr list * expr
32
33
34 type statement =
35     Block of statement list
36   | Expr of expr
37   | VarDecl of varDecl
38   | Return of expr
39   | For of varDecl * expr * varDecl * statement
40   | While of expr * statement
41   | If of expr * statement * statement
42   | Exit
```

```

43     | Break
44     | ForEachLoop of string * string * statement * loopType
45     | Continue
46
47 type functionDefinition = {
48     fname: string;
49     formals: string list;
50     body: statement list
51 }
52
53 type program = statement list * functionDefinition list
54
55 (Supported Types)
56 type builtInType =
57     | Void (For things that don't return anything)
58     | Int
59     | Bool
60     | Mat of int
61     | Annotation of string
62     | Func
63     (returnType * formalType list)
64     | FuncSignature of builtInType * builtInType list
65     | Empty
66     | Keyword
67
68 (Annotated Expression)
69 type aexpr =
70     ALiteral of int * builtInType
71     | ABoolLit of bool * builtInType
72     | AId of string * builtInType
73     | AUnboundedAccessRead of string * builtInType * aexpr * builtInType
74     | AUnboundedAccessWrite of string * builtInType * aexpr * aexpr * builtInType
75     | AMatPlus of string * builtInType * string * builtInType * builtInType
76     | AMatMinus of string * builtInType * string * builtInType * builtInType
77     | AMatAccess of string * builtInType * aexpr list * int * builtInType
78     | ABinaryOp of aexpr * op * aexpr * builtInType
79     | AUnop of uop * aexpr * builtInType
80     | ACall of string * builtInType * aexpr list * builtInType
81     | ANoexpr of builtInType
82
83
84 (Annotated variable declarations)
85 type avarDecl =
86     | AMatrix of string * builtInType * aexpr list list * int * int * builtInType

```

```

87 | AExprAssign of string * builtInType * aexpr * builtInType
88 | ADimAssign of string * builtInType * aexpr list * int * builtInType
89 | AMatElementAssign of string * builtInType * aexpr list * aexpr * int * builtInType
90 | ANodecl of builtInType
91
92
93 (* Annotated statements *)
94 type astatement =
95 | ABlock of astatement list * builtInType
96 | AExpr of aexpr * builtInType
97 | AVarDecl of avarDecl * builtInType
98 | AReturn of builtInType * aexpr * builtInType
99 | AFor of avarDecl * aexpr * avarDecl * astatement * builtInType
100 | AWhile of aexpr * astatement * builtInType
101 | AIf of aexpr * astatement * astatement * builtInType
102 | AExit of builtInType
103 | ABreak of builtInType
104 | AForEachLoop of string * builtInType * string * builtInType * astatement * loop
105 | AContinue of builtInType
106
107 (* Annotated functions *)
108
109 type afunctionDefinition = {
110   afname: string * builtInType;
111   aformals: (string * builtInType) list;
112   abody: astatement list;
113   retType: builtInType
114 }
115
116
117 (* Pretty Printing function *)
118
119 let rec string_of_builtInType = function
120 | Void -> "Void"
121 | Int -> "Int"
122 | Bool -> "Bool"
123 | Mat(nDims) -> "Mat(" ^ string_of_int nDims ^ ")"
124 | Annotation(m) -> m
125 | Func -> "Func"
126 | Empty -> "Empty"
127 | Keyword -> "Keyword"
128 | FuncSignature(returnType, formalTypeList) -> "FuncSignature: ReturnType: " ^ str
129
130

```

```

131
132
133 let string_of_op = function
134   Add -> "+"
135   | Sub -> "-"
136   | Mul -> "*"
137   | Div -> "/"
138   | Equal -> "=="
139   | Neq -> "!="
140   | Less -> "<"
141   | Leq -> "<="
142   | Greater -> ">"
143   | Geq -> ">="
144   | And -> "&&"
145   | Or -> "||"
146   | Mod -> "%"
147   | Exp -> "^"
148
149 let
150   string_of_uop = function
151     | Neg -> "-"
152     | Not -> "!"
153
154   let string_of_boolLit = function
155     true -> "true"
156     | false -> "false"
157
158
159   let rec string_of_aexpr = function
160     | ALiteral(l, t) -> string_of_int l
161     | ABoolLit(b, t) -> string_of_boolLit b
162     | AId(s, t) -> s
163     | AUnboundedAccessRead(id, _, expr, _) -> "<" ^ id ^ "," ^ string_of_aexpr expr ^ ">"
164     | AUnboundedAccessWrite(id, _, expr1, expr2, _) -> "[[" ^ id ^ "," ^ string_of_aexpr
165     | AMatPlus(id1,_,id2,_,_) -> id1 ^ " +. " ^ id2
166     | AMatMinus(id1,_,id2,_,_) -> id1 ^ " -. " ^ id2
167     | AMatAccess(id, t1, exprLst,i, t2) -> id ^ string_of_dim exprLst
168     | ABinaryOp(e1, o, e2, t) ->
169       string_of_aexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_aexpr e2
170     | AUnop(o, e, t) -> string_of_uop o ^ string_of_aexpr e
171     | ACall(f, t1, e1, t2) ->
172       f ^ "(" ^ String.concat ", " (List.map string_of_aexpr e1) ^ ")"
173     | ANoexpr(t) -> ""
174

```

```

175   and string_of_dim = function
176 | [] -> ""
177 | a::lst -> "[" ^ string_of_aexpr a ^ "]" ^ string_of_dim lst
178
179 let rec string_of_row = function
180 | [] -> ""
181 | [a] -> string_of_aexpr a
182 | a::lst-> string_of_aexpr a ^ "," ^ string_of_row lst
183
184
185 let rec string_of_mat = function
186 | [] -> ""
187 | [a] -> string_of_row a
188 | a::lst-> string_of_row a ^ ";" ^ string_of_mat lst
189
190
191 let rec string_of_avarDecl = function
192 | AMatrix(matrixName, t1, mat, i1, i2, t2) -> "\n" ^ string_of_builtInType t1 ^ "
193 | AExprAssign(id, t1, expr, t2) -> string_of_builtInType t1 ^ " " ^ id ^ " = "
194 | ADimAssign(id, t1, expr, i, t2) -> string_of_builtInType t1 ^ " " ^ id ^ "
195 | AMatElementAssign(s, t1 , aexpr1, aexpr2, i, t2) -> string_of_builtInType t1 ^ "
196 | ANodecl(t) -> ""
197
198 let string_of_loopType = function
199 | Row -> "row"
200 | Ele -> "ele"
201 | Pixel -> "pixel"
202
203
204 let rec string_of_aforDecl = function
205 | AExprAssign(id,t1, expr, t2) -> string_of_builtInType t1 ^ " " ^ id ^ " = " ^ string
206 | AMatrix(matrixName, t1, mat, i1, i2, t2) -> matrixName ^ " = {" ^ string_of_mat mat
207 | ADimAssign(id, t1, expr,i , t2) -> string_of_builtInType t1 ^ " " ^ id ^ " = "
208 | AMatElementAssign(s, t1 , aexpr1, aexpr2, i, t2) -> string_of_builtInType t1 ^ " "
209 | ANodecl(t) -> ""
210
211
212
213
214 let rec string_of_astmt = function
215   ABlock(stmts, t) ->
216     "{\n" ^ String.concat "" (List.map string_of_astmt stmts) ^ "}\n"
217 | AExpr(expr, t) -> string_of_aexpr expr ^ ";\n";
218 | AVarDecl(varDecl, t) -> string_of_avarDecl varDecl ^ "\n";

```

```

219 | AReturn(_, expr, t) -> "return " ^ string_of_aexpr expr ^ ";\n";
220 | AIf(e, s1, ABlock([], Void), t) -> "if (" ^ string_of_aexpr e ^ ")\n" ^ string_of
221 | AIf(e, s1, s2, t) -> "if (" ^ string_of_aexpr e ^ ")\n" ^
222     string_of_astmt s1 ^ "else\n" ^ string_of_astmt s2
223 | AFor(v1, e2, v2, s, t) ->
224     "for (" ^ string_of_afortDecl v1 ^ ";\n" ^ string_of_aexpr e2 ^ ";\n" ^
225     string_of_afortDecl v2 ^ ")\n" ^ string_of_astmt s
226 | AWhile(e, s, t) -> "while (" ^ string_of_aexpr e ^ ")\n" ^ string_of_astmt s
227 | AForEachLoop(str1,t1,str2,t2,s1,loopT,t3) -> "for " ^ string_of_loopType loopT ^
228 | AExit(t) -> "exit; \n"
229 | ABreak(t) -> "break; \n"
230 | AContinue(t) -> "continue; \n"
231
232
233
234
235 let rec string_of_aformals = function
236 | [] -> ""
237 | [(s,t)] -> string_of_builtInType t ^ " " ^ s
238 | (s,t)::lst-> string_of_builtInType t ^ " " ^ s ^ ", " ^ string_of_aformals lst
239
240
241
242 let string_of_afname = function
243
244 | (s,t) -> string_of_builtInType t ^ "_" ^ s
245
246
247 let string_of_afdecl fdecl =
248     string_of_builtInType fdecl.retType ^ " " ^ string_of_afname fdecl.afname ^ "(" ^ (
249     ")\n{\n" ^
250     String.concat "" (List.map string_of_astmt fdecl.abody) ^
251     "}\n"
252
253
254 let string_of_program (stmt, funcs) =
255     String.concat "" (List.map string_of_astmt stmt) ^ "\n" ^
256     String.concat "\n" (List.map string_of_afdecl funcs)

```

## 8.4 semant.ml

```

1 (* MatCV Semantic Checker *)
2
3 open Ast

```

```

4
5 (*module ReservedWords = Set.Make(String)*)
6
7 let keywords = ["row"; "col"; "ele"; "pixel"; "var"; "const"; "if";
  ↪ "else"; "for"; "break"; "continue"; "exit"; "while"; "return";
  ↪ "function"; "true"; "false"]
8
9 let builtInFunctions = ["print"; "main"]
10
11 let code = ref ([])
12
13 let getListSize lst = List.fold_left (fun acc _ -> acc + 1) 0 lst
14
15 let generateTypeForAnnotation() =
16 let rec genHelp = function
17 | 'z'::tail -> 'a' :: List.rev (genHelp (List.rev tail))
18 | c::tail -> (Char.chr ((Char.code c) + 1)) :: tail
19 | [] -> ['a']
20 in
21 let updatedCode = genHelp !code
22 in code := updatedCode;
23 Annotation(String.concat "" (List.map Char.escaped updatedCode));;
24
25
26 let typeOfAexpr = function
27 | ALiteral(_, t) -> t
28 | ABoolLit(_, t) -> t
29 | AId(_, t) -> t
30 | AMatPlus (_, _, _, _, t) | AMatMinus (_, _, _, _, t) -> t
31 | AUnboundedAccessRead( _, _, _, t) -> t
32 | AUnboundedAccessWrite( _, _, _, _, t) -> t
33 | AMatAccess( _, _, _, _, t) -> t
34 | ABinaryOp( _, _, _, t) -> t
35 | AUnop( _, _, t) -> t
36 | ACall( _, _, _, t) -> t
37 | ANoexpr(t) -> t
38
39
40 let typeOfAvarDecl = function
41 | ANodecl(t) -> t
42 | AMatrix( _, _, _, _, t) -> t
43 | AExprAssign( _, _, _, t) -> t
44 | ADimAssign( _, _, _, _, t) -> t
45 | AMatElementAssign( _, _, _, _, t) -> t

```



```

46
47
48
49 let typeOfStatement = function
50 | ABlock(_, t) -> t
51 | AExpr(_, t) -> t
52 | AVarDecl(_, t) -> t
53 | AReturn(_, _, t) -> t
54 | AFor(_, _, _, _, t) -> t
55 | AWhile(_, _, t) -> t
56 | AIf(_, _, _, t) -> t
57 | AExit(t) -> t
58 | ABreak(t) -> t
59 | AForEachLoop(_, _, _, _, _, t) -> t
60 | AContinue(t) -> t
61
62 (* Old Code *)
63 let getVariableDeclFromStatement statements =
64     let rec helper acc = function
65     | [] -> acc
66     | VarDecl(s) :: t -> helper (VarDecl(s)::acc) t
67     | _ :: t -> helper acc t
68     in helper [] statements
69
70
71 (* Handle errors *)
72 let printError message =
73     print_string("\nError: " ^ message); exit 1
74
75 let printWarning message =
76     print_string ("\nWarning: " ^ message)
77
78 let printReservedError name = printError ("Name: " ^ name ^ " is
79     ↪ reserved.")
80
81 let printDuplicateFunctionError typ m =
82     match typ with
83     | Keyword -> printReservedError m
84     | _ -> printError ("Multiple definitions of function: " ^
85     ↪ m)
86
87 let printUndefinedVariableError m = printError ("Undefined variable: "
88     ↪ ^ m)

```

```

86 let printInvalidDimensionsError m = printError ("Invalid dimensions
    ↪ were specified for Matrix: " ^ m)
87
88 let printTypeMismatchError id t1 t2 = printError ("Type Mismatch:
    ↪ Cannot assign type " ^ (string_of_builtInType t2) ^ " to " ^ id ^
    ↪ ". Previously had type: " ^ (string_of_builtInType t1) )
89
90
91 let rec annotateExpression globalSymbolTable localSymbolTable =
    ↪ function
92 | Literal(l) -> ALiteral(l, Int)
93 | BoolLit(b) -> ABoolLit(b, Bool)
94 | Id(id) -> let idType = if Hashtbl.mem localSymbolTable id
95                 then Hashtbl.find localSymbolTable id
96                 else if Hashtbl.mem globalSymbolTable id
97                 then Hashtbl.find globalSymbolTable id
98                 else let _ = printUndefinedVariableError id
    ↪ in Void in
    AId(id, idType)
99
100 | MatPlus(id1, id2) ->
101
102     let idType1 = if Hashtbl.mem
    ↪ localSymbolTable id1
103     then Hashtbl.find localSymbolTable id1
104     else if Hashtbl.mem globalSymbolTable id1
105     then Hashtbl.find globalSymbolTable id1
106     else let _ = printUndefinedVariableError id1
    ↪ in Void in
107
108     let idType2 = if Hashtbl.mem
    ↪ localSymbolTable id2
109     then Hashtbl.find localSymbolTable id2
110     else if Hashtbl.mem globalSymbolTable id2
111     then Hashtbl.find globalSymbolTable id2
112     else let _ = printUndefinedVariableError id2
    ↪ in Void in
    AMatPlus(id1, idType1, id2, idType2,
    ↪ generateTypeForAnnotation())
113
114 | MatMinus(id1, id2) ->
115
116     let idType1 = if Hashtbl.mem
    ↪ localSymbolTable id1
117     then Hashtbl.find localSymbolTable id1
118     else if Hashtbl.mem globalSymbolTable id1
    then Hashtbl.find globalSymbolTable id1

```

```

119     else let _ = printUndefinedVariableError id1
120           ↪ in Void in
121
122     let idType2 = if Hashtbl.mem
123                   ↪ localSymbolTable id2
124     then Hashtbl.find localSymbolTable id2
125     else if Hashtbl.mem globalSymbolTable id2
126     then Hashtbl.find globalSymbolTable id2
127     else let _ = printUndefinedVariableError id2
128           ↪ in Void in
129     AMatMinus(id1, idType1, id2, idType2,
130              ↪ generateTypeForAnnotation())
131
132 | UnboundedAccessRead(id, expr) -> let idType = if Hashtbl.mem
133   ↪ localSymbolTable id
134     then Hashtbl.find localSymbolTable id
135     else if Hashtbl.mem globalSymbolTable id
136     then Hashtbl.find globalSymbolTable id
137     else let _ = printUndefinedVariableError id
138           ↪ in Void in
139     let aexpr = annotateExpression
140           ↪ globalSymbolTable localSymbolTable expr
141           ↪ in
142     AUnboundedAccessRead(id, idType, aexpr,
143                          ↪ generateTypeForAnnotation())
144
145 | UnboundedAccessWrite(id, expr1, expr2) -> let idType = if
146   ↪ Hashtbl.mem localSymbolTable id
147     then Hashtbl.find localSymbolTable id
148     else if Hashtbl.mem globalSymbolTable id
149     then Hashtbl.find globalSymbolTable id
150     else let _ = printUndefinedVariableError id
151           ↪ in Void in
152     let aexpr1 = annotateExpression
153           ↪ globalSymbolTable localSymbolTable expr1
154           ↪ in
155     let aexpr2 = annotateExpression
156           ↪ globalSymbolTable localSymbolTable expr2
157           ↪ in
158     AUnboundedAccessWrite(id, idType, aexpr1,
159                           ↪ aexpr2, generateTypeForAnnotation())
160
161 | MatAccess(id, exprList) -> let idType =
162                               if Hashtbl.mem localSymbolTable id then

```

```

147         Hashtbl.find localSymbolTable id
148     else if Hashtbl.mem globalSymbolTable id
149         ↪ then
150         Hashtbl.find globalSymbolTable id
151     else let _ = printUndefinedVariableError
152         ↪ id in Void in
153     let aExprList = List.map (fun expr ->
154         ↪ annotateExpression globalSymbolTable
155         ↪ localSymbolTable expr) exprList in
156     let nDimensions = getListSize exprList
157     ↪ in
158     AMatAccess(id, idType, aExprList,
159     ↪ nDimensions,
160     ↪ generateTypeForAnnotation())
161
162 | BinaryOp(expr1, op, expr2) -> let aexpr1 = annotateExpression
163     ↪ globalSymbolTable localSymbolTable expr1 in
164     let aexpr2 = annotateExpression
165     ↪ globalSymbolTable
166     ↪ localSymbolTable expr2 in
167     ABinaryOp(aexpr1, op, aexpr2,
168     ↪ generateTypeForAnnotation())
169
170 | Unop(uop, expr) -> let aexpr = annotateExpression globalSymbolTable
171     ↪ localSymbolTable expr in
172     AUnop(uop, aexpr,
173     ↪ generateTypeForAnnotation())
174
175 | Call(id, exprList) -> let idType =
176     if Hashtbl.mem localSymbolTable id then
177     Hashtbl.find localSymbolTable id
178     else if Hashtbl.mem globalSymbolTable id then
179     Hashtbl.find globalSymbolTable id
180     else let _ = printUndefinedVariableError id
181     ↪ in Void in
182     let aExprList = List.map (fun expr ->
183     ↪ annotateExpression globalSymbolTable
184     ↪ localSymbolTable expr) exprList in
185     ACall(id, idType, aExprList,
186     ↪ generateTypeForAnnotation())
187
188 | Noexpr -> ANoexpr(Void)

```

```

174
175 let rec annotateVarDecl globalSymbolTable localSymbolTable = function
176   | Nodecl -> ANodecl(Void)
177   | Matrix(id, exprListList) -> let idType =
178     if Hashtbl.mem localSymbolTable id then
179     Hashtbl.find localSymbolTable id
180     else if Hashtbl.mem globalSymbolTable
181       → id then
182     Hashtbl.find globalSymbolTable id
183     else let idt =
184       → generateTypeForAnnotation() in
185     (* Add a generated type to the local
186       → symbol table *)
187     let _ = Hashtbl.add localSymbolTable id
188       → idt
189     in idt
190     in
191     let _ = if idType = Keyword then
192       → printReservedError id
193     in
194     let nRows = getListSize exprListList
195     in
196     let nCols = if nRows <> 0 then
197       getListSize (List.hd exprListList)
198       else 0
199     in
200     (* Check whether all rows have equal
201       → number of elements *)
202     let _ = List.iter (fun exprList -> if
203       → (getListSize exprList) <> nCols
204       → then
205         printInvalidDimensionsError id)
206         → exprListList
207     in
208     (* Annotate each element *)
209     let aExprListList = List.map
210     (fun exprList -> List.map (fun expr ->
211       → annotateExpression
212       → globalSymbolTable
213       → localSymbolTable
214       → expr)

```

```

203                                     exprList)
                                         ↪ exprListList
                                         ↪ in
204     (* Store the row and column count with
       ↪ this matrix *)
205     (* Will help in code generation *)
206     AMatrix(id, idType, aExprListList,
       ↪ nRows, nCols, Void)
207
208 | ExprAssign(id, expr) -> let idType =
209     if Hashtbl.mem localSymbolTable id then
210     Hashtbl.find localSymbolTable id
211     else if Hashtbl.mem globalSymbolTable
       ↪ id then
212     Hashtbl.find globalSymbolTable id
213     else let idt =
       ↪ generateTypeForAnnotation() in
214     (* Add a generated type to the local
       ↪ symbol table *)
215     let _ = Hashtbl.add localSymbolTable id
       ↪ idt
216     in idt
217     in
218     let _ = if idType = Keyword then
       ↪ printReservedError id
219     in
220     let aExpr = annotateExpression
       ↪ globalSymbolTable localSymbolTable
       ↪ expr
221     in
222     AExprAssign(id, idType, aExpr, Void)
223
224 | DimAssign(id, exprList) -> let idType =
225     if Hashtbl.mem localSymbolTable id then
226     Hashtbl.find localSymbolTable id
227     else if Hashtbl.mem globalSymbolTable
       ↪ id then
228     Hashtbl.find globalSymbolTable id
229     else let idt =
       ↪ generateTypeForAnnotation() in
230     (* Add a generated type to the local
       ↪ symbol table *)
231     let _ = Hashtbl.add localSymbolTable id
       ↪ idt

```

```

232     in idt
233     in
234     let _ = if idType = Keyword then
                ↪ printReservedError id
235     in
236     let aExprList = List.map (fun expr ->
237
                ↪ annotateExpression
                ↪ globalSymbolTable
                ↪ localSymbolTable
                ↪ expr)
                exprList
238
239     in
240     let nDimensions = getListSize exprList
                ↪ in
241     ADimAssign(id, idType, aExprList,
                ↪ nDimensions, Void)
242
243 | MatElementAssign(id, exprList, expr) -> let idType =
244     if Hashtbl.mem localSymbolTable id then
245     Hashtbl.find localSymbolTable id
246     else if Hashtbl.mem globalSymbolTable
                ↪ id then
247     Hashtbl.find globalSymbolTable id
248     else
249     let _ = printUndefinedVariableError id
                ↪ in
250     Empty
251     in
252     let _ = if idType = Keyword then
                ↪ printReservedError id
253     in
254     let aExprList = List.map (fun expr ->
255
                ↪ annotateExpression
                ↪ globalSymbolTable
                ↪ localSymbolTable
                ↪ expr)
                exprList
256
257     in
258     let aExpr = annotateExpression
                ↪ globalSymbolTable localSymbolTable
                ↪ expr
259     in

```

```

260         let nDimensions = getListSize exprList
           ↪ in
261         AMatElementAssign(id, idType,
           ↪ aExprList, aExpr, nDimensions,
           ↪ Void)
262
263
264 let mergeSymbolTables globalSymbolTable localSymbolTable = let
  ↪ mergedSymbolTable = Hashtbl.create 100 in
265     let _ = Hashtbl.iter (fun key value -> Hashtbl.add
  ↪ mergedSymbolTable key value) localSymbolTable
266     in
267     let _ = Hashtbl.iter (fun key value -> if not (Hashtbl.mem
  ↪ mergedSymbolTable key) then Hashtbl.add mergedSymbolTable
  ↪ key value) globalSymbolTable
268     in mergedSymbolTable
269
270
271
272
273 let rec annotateStatement globalSymbolTable localSymbolTable
  ↪ ?isControlFlowAllowed:(isCFA = false) inFunction = function
274 | Block (statementList) -> let newGlobalSymbolTable =
  ↪ mergeSymbolTables globalSymbolTable localSymbolTable in
275     let newLocalSymbolTable = Hashtbl.create
  ↪ 100 in
276     let aStatementList = List.map (fun
  ↪ statement -> annotateStatement
  ↪ newGlobalSymbolTable
  ↪ newLocalSymbolTable inFunction
  ↪ statement
  ↪ ~isControlFlowAllowed:isCFA)
  ↪ statementList
277     in
278     ABlock (aStatementList, Void)
279 | Expr (expr) -> let aExpr = annotateExpression globalSymbolTable
  ↪ localSymbolTable expr in
280     AExpr(aExpr, Void)
281 | VarDecl (varDecl) -> let aVarDecl = annotateVarDecl
  ↪ globalSymbolTable localSymbolTable varDecl in
282     AVarDecl(aVarDecl, Void)
283 | Return (expr) -> let _ = if inFunction = "main" then printError
  ↪ "Cannot use return outside functions." in let aExpr =
  ↪ annotateExpression globalSymbolTable localSymbolTable expr in

```



```

284 let funcType = Hashtbl.find globalSymbolTable inFunction
285 in (match funcType with
286 | FuncSignature(returnTypeSig, formalTypeList) -> AReturn
  ↪ (returnTypeSig, aExpr, Void)
287 | _ -> let _ = printError "Invalid use of return statement." in
  ↪ AReturn (Void, aExpr, Void)
288 )
289
290 | For (varDecl1, expr, varDecl2, statement) -> let aVarDecl1 =
  ↪ annotateVarDecl globalSymbolTable localSymbolTable varDecl1 in
291                                     let aVarDecl2 =
  ↪ annotateVarDecl
  ↪ globalSymbolTable
  ↪ localSymbolTable
  ↪ varDecl2 in
292                                     let aExpr =
  ↪ annotateExpression
  ↪ globalSymbolTable
  ↪ localSymbolTable
  ↪ expr in
293                                     let aStatement =
  ↪ annotateStatement
  ↪ globalSymbolTable
  ↪ localSymbolTable
  ↪ ~isControlFlowAllowed:true
  ↪ inFunction
  ↪ statement in
294                                     AFor (aVarDecl1,
  ↪ aExpr,
  ↪ aVarDecl2,
  ↪ aStatement,
  ↪ Void)
295 | While (expr, statement) -> let aExpr = annotateExpression
  ↪ globalSymbolTable localSymbolTable expr in
296                                     let aStatement = annotateStatement
  ↪ globalSymbolTable localSymbolTable
  ↪ ~isControlFlowAllowed:true
  ↪ inFunction statement in
297                                     AWhile (aExpr, aStatement, Void)
298 | If (expr, statement1, statement2) ->
299     let newGlobalSymbolTable = mergeSymbolTables
  ↪ globalSymbolTable localSymbolTable in
300     let newLocalSymbolTable = Hashtbl.create 100 in

```

```

301     let aExpr = annotateExpression newGlobalSymbolTable
        ↪ newLocalSymbolTable expr in
302         let aStatement1 =
            ↪ annotateStatement
            ↪ newGlobalSymbolTable
            ↪ newLocalSymbolTable
            ↪ ~isControlFlowAllowed:isCFA
            ↪ inFunction statement1 in
303         let aStatement2 =
            ↪ annotateStatement
            ↪ newGlobalSymbolTable
            ↪ newLocalSymbolTable
            ↪ ~isControlFlowAllowed:isCFA
            ↪ inFunction statement2 in
304         AIf(aExpr, aStatement1,
            ↪ aStatement2, Void)
305 | Exit -> AExit(Void)
306 | Break -> let _ = if not isCFA then printError "Invalid use of
        ↪ break." in ABreak(Void)
307 | ForEachLoop (id, objName, statement, loopType) ->
308     let newGlobalSymbolTable = mergeSymbolTables
        ↪ globalSymbolTable localSymbolTable in
309     let newLocalSymbolTable = Hashtbl.create 100 in
310     let idType =
311         if Hashtbl.mem newLocalSymbolTable id then
312             Hashtbl.find newLocalSymbolTable id
313         else if Hashtbl.mem newGlobalSymbolTable id
            ↪ then
314             Hashtbl.find newGlobalSymbolTable id
315         else let idt = generateTypeForAnnotation() in
316             (* Add a generated type to the local symbol
            ↪ table *)
317             let _ = Hashtbl.add newLocalSymbolTable id idt
318             in idt
319         in
320         let objType =
321             if Hashtbl.mem newLocalSymbolTable objName then
322                 Hashtbl.find newLocalSymbolTable objName
323             else if Hashtbl.mem newGlobalSymbolTable
            ↪ objName then
324                 Hashtbl.find newGlobalSymbolTable objName
325             else
326                 let _ = printUndefinedVariableError objName in
327                 Empty

```

```

328         in
329         let aStatement = annotateStatement
           ↪ newGlobalSymbolTable newLocalSymbolTable
           ↪ ~isControlFlowAllowed:true inFunction
           ↪ statement
330     in
331     AForEachLoop (id, idType, objName, objType,
           ↪ aStatement, loopType, Void)
332 | Continue -> let _ = if not isCFA then printError "Invalid use of
           ↪ continue." in AContinue(Void)
333
334
335 let rec collectExpr = function
336 | ALiteral(_) | ABoolLit(_) | AId(_) | ANoexpr(_) -> []
337 (* If someone accesses a variable like matrix, it means that id's
338 * type should be Mat and each expression should evaluate to Int and
339 * this expression returns an Int *)
340 | AUnboundedAccessRead(id, idType, aexpr, exprType) -> let
           ↪ constraints = [(exprType, Int)]
341 in let exprConstr = collectExpr aexpr in [(typeOfAexpr aexpr, Int)] @
           ↪ constraints @ exprConstr
342
343 | AUnboundedAccessWrite(id, idType, aexpr1, aexpr2, exprType) ->
           ↪ let constraints = [(exprType, Int)]
344 in let exprConstr1 = collectExpr aexpr1
345 in let exprConstr2 = collectExpr aexpr2 in [(typeOfAexpr aexpr1,
           ↪ Int);(typeOfAexpr aexpr2, Int)] @ constraints @ exprConstr1 @
           ↪ exprConstr2
346
347 | AMatPlus(id1, idType1, id2, idType2, exprType) -> [(exprType,
           ↪ idType1); (idType1, idType2)]
348
349 | AMatMinus(id1, idType1, id2, idType2, exprType) -> [(exprType,
           ↪ idType1); (idType1, idType2)]
350
351 | AMatAccess(id, idType, aExprList, nDim, exprType) ->
352     let constraints = [(idType, Mat(nDim)); (exprType, Int)] (*
           ↪ Not supporting a matrix of functions for now *)
353     in let exprConstraints = List.fold_left (fun constraintAcc
           ↪ expr -> let exprConstr = collectExpr expr in (typeOfAexpr
           ↪ expr, Int) :: exprConstr @ constraintAcc) [] aExprList
354     in constraints @ exprConstraints
355 (* Now in case of binary operators, the result can be bool if the
356 * operators are comparison operators etc. *)

```

```

357 | ABinaryOp(aexpr1, op, aexpr2, exprType) ->
358     let t1 = typeOfAexpr aexpr1 in let t2 = typeOfAexpr aexpr2
359     in
360     let constraints = match op with
361     | Equal | Neq | Less | Leq | Greater | Geq | And | Or ->
362         [(t1, t2); (exprType, Bool)]
363     | Add | Sub | Mul | Div | Exp | Mod -> [(t1, Int); (t2, Int)];
364     ↪ (exprType, Int)]
365     in
366     constraints @ (collectExpr aexpr1) @ (collectExpr aexpr2)
367
368 | AUnop(uop, aexpr, exprType) ->
369     let t = typeOfAexpr aexpr in
370     let constraints = match uop with
371     | Neg -> [(t, Int); (exprType, Int)]
372     | Not -> [(exprType, Bool)]
373     in
374     constraints @ (collectExpr aexpr)
375
376     (* TODO: Add more constraints using function definition: *)
377 | ACall(id, idType, aExprList, exprType) ->
378     match idType with
379     | FuncSignature(returnTypeSig, formalTypeList) ->
380     let exprConstraints = List.fold_left (fun constraintAcc expr
381     ↪ -> let exprConstr = collectExpr expr in exprConstr @
382     ↪ constraintAcc) [] aExprList
383     in let size1 = getListSize formalTypeList in let size2 =
384     ↪ getListSize aExprList in
385     let _ = if size1 != size2 then printError ("Function: " ^ id
386     ↪ ^ " called with: " ^ string_of_int size2 ^ " arguments.
387     ↪ While the function expects: " ^ string_of_int size1 ^ "
388     ↪ arguments.")
389     in let formalConstraints = List.map2 (fun typ1 aExpr ->
390     ↪ (typ1, typeOfAexpr(aExpr))) formalTypeList aExprList
391     in let retConstraint = [(returnTypeSig, exprType)]
392     in retConstraint @ formalConstraints @ exprConstraints
393     | Keyword when id = "print" -> []
394     | _ -> let _ = printError "Invalid use of function call." in
395     ↪ []
396
397
398
399
400 let rec collectVarDecl = function
401     | ANodecl(_) -> []

```

```

392 | AMatrix(id, idType, aExprListList, nRows, nCols, _) ->
393     let constraints = [(idType, Mat(2))]
394     in
395     let aExprListListConstraints = List.fold_left (fun
396         ↪ constraintList aExprList ->
397         List.fold_left (fun constrList aexpr -> (typeOfAexpr
398             ↪ aexpr, Int) ::(collectExpr aexpr) @ constrList)
399             ↪ constraintList aExprList
400         ) [] aExprListList
401     in constraints @ aExprListListConstraints
402
403 | AExprAssign(id, idType, aExpr, _) ->
404     let constraints = [(idType, typeOfAexpr aExpr)]
405     in
406     (collectExpr aExpr) @ constraints
407
408 | ADimAssign(id, idType, aExprList, nDimensions, _) ->
409     let constraints = [(idType, Mat(nDimensions))]
410     in
411     let aExprListConstraints = List.fold_left (fun constrList
412         ↪ expr -> (typeOfAexpr expr, Int) ::(collectExpr expr) @
413         ↪ constrList) [] aExprList
414     in constraints @ aExprListConstraints
415
416 | AMatElementAssign(id, idType, aExprList, aExpr, nDimensions, _)
417     ↪ ->
418     let constraints = [(idType, Mat(nDimensions)); (typeOfAexpr
419         ↪ aExpr, Int)]
420     in
421     let aExprListConstraints = List.fold_left (fun constrList
422         ↪ expr -> (typeOfAexpr expr, Int) ::(collectExpr expr) @
423         ↪ constrList) [] aExprList
424     in constraints @ aExprListConstraints @ (collectExpr aExpr)
425
426     (* All statements have type Void *)
427 let rec collectStatement = function
428 | AContinue(_) | ABreak(_) | AExit(_) -> []
429
430 | ABlock (aStatementList, _) -> List.fold_left (fun constraintAcc
431     ↪ astatement -> (collectStatement astatement) @ constraintAcc) []
432     ↪ aStatementList
433
434

```

```

425 | AExpr(aExpr, _) -> collectExpr aExpr
426
427 | AVarDecl(aVarDecl, _) -> collectVarDecl aVarDecl
428
429 (* TODO: Relate return type to the annotated function *)
430 | AReturn (retType, aExpr, _) -> [(retType, typeOfAexpr(aExpr))] @
    ↪ (collectExpr aExpr)
431
432 | AFor (aVarDecl1, aExpr, aVarDecl2, aStatement, _) -> let
    ↪ constLst1 = collectVarDecl aVarDecl1
433
434
435
436
437
438
439
440
441
442 | AIf(aExpr, aStatement1, aStatement2, _) -> let constraints =
    ↪ (typeOfAexpr aExpr, Bool) :: (collectExpr aExpr)
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

444                                     let constLst1 =
                                        ↪ collectStatement
                                        ↪ aStatement1
445                                     in
446                                     let constLst2 =
                                        ↪ collectStatement
                                        ↪ aStatement2
447                                     in
448                                     constraints @
                                        ↪ constLst1 @
                                        ↪ constLst2
449
450 | AWhile (aExpr, aStatement, _) -> let constraints = (typeOfAexpr
451 ↪ aExpr, Bool) :: (collectExpr aExpr)
452                                     in
453                                     let constLst = collectStatement
                                        ↪ aStatement
454                                     in constraints @ constLst
455 | AForEachLoop (id, idType, objName, objType, aStatement, loopType,
456 ↪ _) -> let constraints = match loopType with
457                                     | Row -> [(idType, Mat(2));
                                        ↪ (objType, Mat(3))]
458                                     | Ele -> [(idType, Int);
                                        ↪ (objType, Mat(3))]
459                                     | Pixel -> [(idType, Mat(1));
                                        ↪ (objType, Mat(3))]
460                                     in
                                        constraints @ (collectStatement
                                        ↪ aStatement)
461
462
463 let rec substitute t1 t2 t =
464     match t with
465     | Void | Int | Bool | Func -> t
466     | Mat(nDim) -> Mat(nDim)
467     | Annotation(s) -> if t1 = t then t2 else t
468     | FuncSignature(_) -> Func
469     | Keyword -> Keyword
470     | _ -> printError "Unknown type error."
471
472 let apply substitutionList typ =
473     List.fold_right (fun (t1, t2) t -> substitute t1 t2 t)
        ↪ substitutionList typ

```

```

474
475
476 let rec unifyOne s t =
477   if s = t then (*let _ = print_string ("Unify one s = t:" ^
478     ↪ string_of_builtInType s ^ "," ^ string_of_builtInType t ^ "\n")
479     ↪ in*) []
480   else
481     match (s, t) with
482     | Annotation(x), Annotation(y) -> [Annotation(x), Annotation(y)]
483     | Annotation(x), y | y, Annotation(x) -> [(Annotation(x), y)]
484     | x , y -> let _ = printError ("Mismatched types:" ^
485       ↪ string_of_builtInType x ^ "," ^ string_of_builtInType y ^
486       ↪ "\n") in []
487 and unify = function
488 | [] -> []
489 | (x, y) :: t ->
490   let t2 = unify t in
491   let t1 = unifyOne (apply t2 x) (apply t2 y) in
492   t1 @ t2
493 let collectStatementList astatements = let constraints = List.fold_left
494   ↪ (fun constraintList astatement -> (constraintList @
495   ↪ (collectStatement astatement))) [] astatements in constraints
496
497 let annotateFunctionHelper globalSymbolTable localSymbolTable func =
498   let funcType = Hashtbl.find globalSymbolTable func.fname
499   in match funcType with
500   | FuncSignature(returnTypeSig, formalTypeList) ->
501     let aFormals = List.map2 (fun id typ -> let _ = (if
502       ↪ Hashtbl.mem localSymbolTable id then printError ("Two
503       ↪ or more formals have same name: " ^ id ^ " in function:
504       ↪ " ^ func.fname)) in let _ = Hashtbl.add
505       ↪ localSymbolTable id typ in (id,typ)) func.formals
506     ↪ formalTypeList
507   in
508   {
509     afname = (func.fname, funcType);
510     aformals = aFormals;
511     abody = List.map (fun statement -> annotateStatement
512       ↪ globalSymbolTable localSymbolTable func.fname statement)
513     ↪ func.body;
514     retType = returnTypeSig;
515   }

```



```

505 | _ -> let _ = printError "Incorrect use of function: " ^
    ↪ func.fname in
506 (* Record shown below is useless. It is here to allow printError
    ↪ to work *)
507 {
508     afname = (func.fname, Void);
509     aformals = List.map (fun id -> let _ = Hashtbl.add
    ↪ localSymbolTable id Void in (id,Void)) func.formals;
510     abody = List.map (fun statement -> annotateStatement
    ↪ globalSymbolTable localSymbolTable func.fname statement)
    ↪ func.body;
511     retType = Void;
512 }
513
514
515
516 let annotateFunction globalSymbolTable func =
517     let localSymbolTable = Hashtbl.create 100
518     in
519     annotateFunctionHelper globalSymbolTable localSymbolTable func
520
521
522
523 let collectFunction func = collectStatementList func.abody
524 let collectFunctionList functions = let constraints = List.fold_left
    ↪ (fun constraintList func -> (constraintList @ (collectFunction
    ↪ func))) [] functions in constraints
525
526
527 let rec applyExpression unifiedConstraints = function
528 | ALiteral(_) as x -> x | ABoolLit(_) as x -> x | ANoexpr(_) as x ->
    ↪ x
529 | AId(id, idType) -> AId(id, (apply unifiedConstraints idType))
530 | AUnboundedAccessRead(id, idType, aexpr, exprType) ->
    ↪ AUnboundedAccessRead(id, (apply unifiedConstraints idType),
    ↪ (applyExpression unifiedConstraints aexpr), (apply
    ↪ unifiedConstraints exprType))
531 | AUnboundedAccessWrite(id, idType, aexpr1, aexpr2, exprType) ->
    ↪ AUnboundedAccessWrite(id, (apply unifiedConstraints idType),
    ↪ (applyExpression unifiedConstraints aexpr1), (applyExpression
    ↪ unifiedConstraints aexpr2), (apply unifiedConstraints exprType))
532

```

```

533 | AMatPlus(id1, idType1, id2, idType2, exprType) -> AMatPlus(id1,
    ↪ (apply unifiedConstraints idType1), id2, (apply
    ↪ unifiedConstraints idType2), (apply unifiedConstraints exprType))
534
535 | AMatMinus(id1, idType1, id2, idType2, exprType) -> AMatMinus(id1,
    ↪ (apply unifiedConstraints idType1), id2, (apply
    ↪ unifiedConstraints idType2), (apply unifiedConstraints exprType))
536
537 | AMatAccess(id, idType, aExprList, nDim, exprType) ->
538     let resolvedExprList = List.map (fun aexpr -> applyExpression
    ↪ unifiedConstraints aexpr) aExprList
539     in
540     AMatAccess(id, (apply unifiedConstraints idType),
    ↪ resolvedExprList, nDim, (apply unifiedConstraints
    ↪ exprType))
541 | ABinaryOp(aexpr1, op, aexpr2, exprType) ->
542     ABinaryOp((applyExpression unifiedConstraints aexpr1), op,
    ↪ (applyExpression unifiedConstraints aexpr2), (apply
    ↪ unifiedConstraints exprType))
543
544 | AUnop(uop, aexpr, exprType) -> AUnop(uop, applyExpression
    ↪ unifiedConstraints aexpr, apply unifiedConstraints exprType)
545 | ACall(id, idType, aExprList, exprType) ->
546     let resolvedExprList = List.map (fun aexpr -> applyExpression
    ↪ unifiedConstraints aexpr) aExprList
547     in
548     ACall(id, (apply unifiedConstraints idType),
    ↪ resolvedExprList, (apply unifiedConstraints exprType))
549
550
551
552 let rec applyVarDecl unifiedConstraints = function
553 | ANodecl(_) as x -> x
554 | AMatrix(id, idType, aExprListList, nRows, nCols, varDeclType) ->
555     let resolveExpr = List.map (fun aexpr -> applyExpression
    ↪ unifiedConstraints aexpr)
556     in
557     let resolvedExprListList = List.map (fun aexprList ->
    ↪ resolveExpr aexprList) aExprListList
558     in
559     AMatrix(id, (apply unifiedConstraints idType),
    ↪ resolvedExprListList, nRows, nCols, (apply
    ↪ unifiedConstraints varDeclType))
560 | AExprAssign(id, idType, aExpr, varDeclType) ->

```

```

561     AExprAssign(id, (apply unifiedConstraints idType),
    ↪ (applyExpression unifiedConstraints aExpr), (apply
    ↪ unifiedConstraints varDeclType))
562 | ADimAssign(id, idType, aExprList, nDimensions, varDeclType) ->
563     let resolvedExprList = List.map (fun aexpr -> applyExpression
    ↪ unifiedConstraints aexpr) aExprList
564     in
565     ADimAssign(id, (apply unifiedConstraints idType),
    ↪ resolvedExprList, nDimensions, (apply unifiedConstraints
    ↪ varDeclType))
566 | AMatElementAssign(id, idType, aExprList, aExpr, nDimensions,
    ↪ varDeclType) ->
567     let resolvedExprList = List.map (fun aexpr -> applyExpression
    ↪ unifiedConstraints aexpr) aExprList
568     in
569     AMatElementAssign(id, (apply unifiedConstraints idType),
    ↪ resolvedExprList, (applyExpression unifiedConstraints
    ↪ aExpr), nDimensions, (apply unifiedConstraints
    ↪ varDeclType))
570
571
572 let rec applyStatement unifiedConstraints = function
573 | AContinue(_) as x -> x | ABreak(_) as x -> x | AExit(_) as x -> x
574
575 | ABlock (aStatementList, statementType) ->
576     let resolvedStatementList = List.map (fun astatement ->
    ↪ applyStatement unifiedConstraints astatement)
    ↪ aStatementList
577     in
578     ABlock(resolvedStatementList, (apply unifiedConstraints
    ↪ statementType))
579
580 | AExpr(aExpr, statementType) ->
581     AExpr((applyExpression unifiedConstraints aExpr), (apply
    ↪ unifiedConstraints statementType))
582
583 | AVarDecl(aVarDecl, statementType) ->
584     AVarDecl((applyVarDecl unifiedConstraints aVarDecl), (apply
    ↪ unifiedConstraints statementType))
585
586 | AReturn(returnType, aExpr, statementType) ->
587     AReturn((apply unifiedConstraints returnType),
    ↪ (applyExpression unifiedConstraints aExpr), (apply
    ↪ unifiedConstraints statementType))

```

```

588
589 | AFor(aVarDecl1, aExpr, aVarDecl2, aStatement, statementType) ->
590   AFor((applyVarDecl unifiedConstraints aVarDecl1),
        ↪ (applyExpression unifiedConstraints aExpr), (applyVarDecl
        ↪ unifiedConstraints aVarDecl2), (applyStatement
        ↪ unifiedConstraints aStatement), (apply unifiedConstraints
        ↪ statementType))
591
592
593 | AIf(aExpr, aStatement1, aStatement2, statementType) ->
594   AIf((applyExpression unifiedConstraints aExpr),
        ↪ (applyStatement unifiedConstraints aStatement1),
        ↪ (applyStatement unifiedConstraints aStatement2), (apply
        ↪ unifiedConstraints statementType))
595
596 | AWhile(aExpr, aStatement, statementType) ->
597   AWhile((applyExpression unifiedConstraints aExpr),
        ↪ (applyStatement unifiedConstraints aStatement), (apply
        ↪ unifiedConstraints statementType))
598
599 | AForEachLoop(id, idType, objName, objType, aStatement, loopType,
        ↪ statementType) ->
600   AForEachLoop(id, (apply unifiedConstraints idType), objName,
        ↪ (apply unifiedConstraints objType), (applyStatement
        ↪ unifiedConstraints aStatement), loopType, (apply
        ↪ unifiedConstraints statementType))
601
602
603 let rec applyStatementList unifiedConstraints aStatementList = List.map
        ↪ (fun astatement -> applyStatement unifiedConstraints astatement)
        ↪ aStatementList
604
605
606
607 let applyFunction unifiedConstraints func =
608   let (fname, _) = func.afname in
609   {
610     afname = (fname, Func);
611     aformals = List.map (fun (id, typ) -> (id, (apply
        ↪ unifiedConstraints typ))) func.aformals;
612     abody = applyStatementList unifiedConstraints func.abody;
613     retType = apply unifiedConstraints func.retType (*let rType =
        ↪ (apply unifiedConstraints func.retType) in match rType
        ↪ with

```

```

614         / Annotation(_) -> Void
615         / x -> x*)
616     ;
617 }
618
619 let applyFunctions unifiedConstraints functions =
620     List.map (fun func -> applyFunction unifiedConstraints func)
621     ↪ functions
622
623     (* Check program semantics *)
624 let check_semantics (gstatements, functions) =
625     let globalSymbolTable = Hashtbl.create 100 in
626     let _ =
627         List.iter (fun ele -> Hashtbl.add globalSymbolTable ele
628             ↪ Keyword) (keywords @ builtInFunctions)
629     in
630     (* Check for duplicate functions *)
631     let _ = List.iter (fun ele ->
632         if Hashtbl.mem globalSymbolTable ele.fname then
633             printDuplicateFunctionError (Hashtbl.find globalSymbolTable
634                 ↪ ele.fname) ele.fname
635         else let formalTypes = List.map (fun _ ->
636             ↪ generateTypeForAnnotation()) ele.formals
637         in
638             Hashtbl.add globalSymbolTable ele.fname
639             ↪ (FuncSignature(generateTypeForAnnotation(),
640                 ↪ formalTypes))) functions
641     in
642     let localSymbolTable = Hashtbl.create 100 in
643     let agstatements = List.map (fun statement -> annotateStatement
644         ↪ globalSymbolTable localSymbolTable "main" statement)
645         ↪ gstatements
646     (* Overwrite globalSymbolTable with localSymbolTable *)
647     in let globalSymbolTable = mergeSymbolTables globalSymbolTable
648         ↪ localSymbolTable in
649     let gconstraints = collectStatementList agstatements
650     in
651     let afunctions = List.map (fun func -> annotateFunction
652         ↪ globalSymbolTable func) functions
653     in
654     let fconstraints = collectFunctionList afunctions
655     in
656     let constraints = gconstraints @ fconstraints

```

```

648   in
649   let unifiedConstraints = unify constraints
650   in
651   let resolvedGStatements = applyStatementList unifiedConstraints
     ↪ agstatements
652   in
653   let resolvedFunctions = applyFunctions unifiedConstraints
     ↪ afunctions
654   (*in
655   let _ = print_string(Ast.string_of_program(resolvedGStatements,
     ↪ resolvedFunctions)) *)
656   in
657   resolvedGStatements, resolvedFunctions

```

## 8.5 library.matcv

```

1  function copyMat(srcMat, destMat)
2  {
3      nDims = <srcMat, 0>;
4      matrixSize = 1;
5
6      for (i = 1; i <= nDims; i = i + 1)
7      {
8          matrixSize = matrixSize * <srcMat, i>;
9      }
10
11     matrixSize = matrixSize + 1 + nDims;
12
13     for (i = 0; i < matrixSize; i = i + 1)
14     {
15         [[destMat, i, <srcMat, i>]];
16     }
17
18     srcMat[0];
19     return destMat[0];
20 }
21
22 function addMat(destMat, srcMat)
23 {
24     nDims = <srcMat, 0>;
25     matrixSize = 1;
26
27     for (i = 1; i <= nDims; i = i + 1)
28     {

```

```

29     matrixSize = matrixSize * <srcMat, i>;
30 }
31
32 matrixSize = matrixSize + 1 + nDims;
33
34
35 for (i = nDims + 1; i < matrixSize; i = i + 1)
36 {
37     temp1 = <destMat, i>;
38     temp2 = <srcMat, i>;
39     [[destMat, i, temp1 + temp2]];
40 }
41
42 srcMat[0];
43 destMat[0];
44 return 0;
45 }
46
47 function minusMat(destMat, srcMat)
48 {
49     nDims = <srcMat, 0>;
50     matrixSize = 1;
51
52     for (i = 1; i <= nDims; i = i + 1)
53     {
54         matrixSize = matrixSize * <srcMat, i>;
55     }
56
57     matrixSize = matrixSize + 1 + nDims;
58
59     for (i = nDims + 1; i < matrixSize; i = i + 1)
60     {
61         temp1 = <destMat, i>;
62         temp2 = <srcMat, i>;
63         [[destMat, i, temp1 - temp2]];
64     }
65
66     srcMat[0];
67     destMat[0];
68     return 0;
69 }

```

## 8.6 codegen.ml

```
1  (*
2   * Code generation for MatCV
3   *)
4
5  module L = Llvml
6  module A = Ast
7
8  let printError message =
9      print_string("\nError: " ^ message); exit 1
10
11
12  let printWarning message =
13      print_string ("\nWarning: " ^ message)
14
15
16
17  let mergeSymbolTables globalSymbolTable localSymbolTable = let
18      ↪ mergedSymbolTable = Hashtbl.create 100 in
19      let _ = Hashtbl.iter (fun key value -> Hashtbl.add
20          ↪ mergedSymbolTable key value) localSymbolTable
21      in
22      let _ = Hashtbl.iter (fun key value -> if not (Hashtbl.mem
23          ↪ mergedSymbolTable key) then Hashtbl.add mergedSymbolTable
24          ↪ key value) globalSymbolTable
25      in mergedSymbolTable
26
27
28
29
30
31
32
33
34  let translate (gstatements, functions) =
35      let context = L.global_context () in
36      let the_module = L.create_module context "MatCV"
37      and i32_t = L.i32_type context
38      and i8_t = L.i8_type context
39      and i1_t = L.i1_type context
40      and void_t = L.void_type context
41      and mat_t = L.pointer_type (L.i32_type context) in
42
43      let ltypeOfType = function
44          | A.Int -> i32_t
45          | A.Bool -> i1_t
46          | A.Mat(_) -> mat_t
47          | A.Void -> void_t
```



```

39 | A.Annotation _ -> printError "Unable to resolve symbols"
40 | A.Func | A.FuncSignature (_, _) | A.Empty | A.Keyword -> printError
   ↪ "Error during compilation. Should have caught this in semantic
   ↪ check."
41 in
42
43 let initOfType = function
44 | A.Mat(_) -> (L.const_pointer_null mat_t)
45 | t -> L.const_int (ltypeOfType t) 0
46
47
48 in
49
50
51 let globalSymbolTable = Hashtbl.create 100
52 in
53 let mainaexpr = A.ALiteral(0, A.Int)
54 in let mainreturnstatement = A.AReturn(A.Int, mainaexpr, A.Void)
55 in let gstatements = gstatements @ [(mainreturnstatement)]
56 in
57 (* Generate code for global statements *)
58 let functions = ({A.afname = ("main", A.Func); A.aformals = [] ;
   ↪ A.abody = gstatements; A.retType = A.Int}) :: functions
59 in
60
61 let createGlobalVar id idType symbolTable = let _ = if not
   ↪ (Hashtbl.mem symbolTable id) then
62     let init = initOfType idType
63     in
64     Hashtbl.add symbolTable id ((L.define_global id init
   ↪ the_module), idType) in ()
65
66
67 in
68
69 let declareGlobalVariableUsingStatement symbolTable = function
70 | A.AVarDecl(x, _) -> (match x with
71
72     | A.AMatrix(id, idType, _, _, _, _) -> createGlobalVar id
   ↪ idType symbolTable
73     | A.AExprAssign(id, idType, _, _) -> createGlobalVar id
   ↪ idType symbolTable
74     | A.ADimAssign(id, idType, _, _, _) -> createGlobalVar id
   ↪ idType symbolTable

```

```

75
76         | _ -> ()
77     )
78
79     | _ -> ()
80
81     in
82
83     (* Generate code for declaration of global variables *)
84     let _ = List.iter (declareGlobalVariableUsingStatement
85         ↪ globalSymbolTable) gstatements
86
87     in
88     let functionTable = Hashtbl.create 100
89     in
90
91     let functionDecl afunc =
92         let name = (fst afunc.A.afname)
93         and formals = Array.of_list (List.map (fun (_, typ) ->
94             ↪ ltypeOfTyp typ) afunc.A.aformals)
95         in let ftype = L.function_type (ltypeOfTyp afunc.A.retType)
96             ↪ formals in
97         Hashtbl.add functionTable name (L.define_function name ftype
98             ↪ the_module, afunc)
99     in
100
101     let _ = List.iter functionDecl functions
102
103     in
104
105     (* Generate code for forward declaration of functions *)
106
107     (* TODO: Uncomment the following lines *)
108
109     let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t
110         ↪ |] in
111     let printf_func = L.declare_function "printf" printf_t the_module
112
113     in
114
115     let copyMat_t = L.function_type i32_t [| L.pointer_type i32_t;
116         ↪ L.pointer_type i32_t|] in
117     let copyMat_func = L.declare_function "copyMat" copyMat_t the_module
118
119     in

```

```

113 let minusMat_t = L.function_type i32_t [| L.pointer_type i32_t;
    ↪ L.pointer_type i32_t|] in
114 let minusMat_func = L.declare_function "minusMat" minusMat_t
    ↪ the_module
115 in
116
117 let addMat_t = L.function_type i32_t [| L.pointer_type i32_t;
    ↪ L.pointer_type i32_t|] in
118 let addMat_func = L.declare_function "addMat" addMat_t the_module
119 in
120
121 let generateFunctionBody afunc =
122   let (theFunction, _) = Hashtbl.find functionTable (fst
    ↪ afunc.A.afname)
123   in
124   let builder = L.builder_at_end context (L.entry_block theFunction)
125   in
126   let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder
127
128   in
129   let memoryMap = Hashtbl.create 100
130   in
131   let localSymbolTable = Hashtbl.create 100
132   in
133
134   let declareFormals (name, typ) value =
135     let _ = (L.set_value_name name value;
136     let local = L.build_alloca (ltypeOfType typ) name builder
137     in
138     ignore (L.build_store value local builder);
139     Hashtbl.add localSymbolTable name (local, typ)
140     )
141     in
142     ()
143   in
144   let _ = List.iter2 declareFormals afunc.A.aformals (Array.to_list
    ↪ (L.params theFunction))
145   in
146   let lookup globalSymbolTable localSymbolTable name = if Hashtbl.mem
    ↪ localSymbolTable name then (fst (Hashtbl.find localSymbolTable
    ↪ name))
147     else (fst (Hashtbl.find globalSymbolTable name))
148   in
149

```

```

150
151
152 let rec genCodeForExpression globalSymbolTable localSymbolTable
  ↪ memoryMap builder = function
153   | A.ALiteral(value, _) -> L.const_int i32_t value
154   | A.ABoolLit(value, _) -> L.const_int i1_t (if value then 1 else
  ↪ 0)
155   | A.AId(id, _) -> L.build_load (lookup globalSymbolTable
  ↪ localSymbolTable id) id builder
156   | A.AUnboundedAccessRead(id, idType, aexpr, _) -> let
  ↪ loadedLMatrix = L.build_load (lookup globalSymbolTable
  ↪ localSymbolTable id) id builder
157   in let e = genCodeForExpression globalSymbolTable
  ↪ localSymbolTable memoryMap builder aexpr
158   in
159   let iPtr = (L.build_in_bounds_gep loadedLMatrix [|e|] (id ^
  ↪ "_iPtr") builder)
160   in
161   L.build_load iPtr id builder
162
163   | A.AUnboundedAccessWrite(id, idType, aexpr1, aexpr2, _) -> let
  ↪ loadedLMatrix = L.build_load (lookup globalSymbolTable
  ↪ localSymbolTable id) id builder
164   in let e1 = genCodeForExpression globalSymbolTable
  ↪ localSymbolTable memoryMap builder aexpr1
165   in let e2 = genCodeForExpression globalSymbolTable
  ↪ localSymbolTable memoryMap builder aexpr2
166   in
167   let iPtr = (L.build_in_bounds_gep loadedLMatrix [|e1|] (id ^
  ↪ "_iPtr") builder)
168   in
169   let _ = L.build_store e2 iPtr builder
170   in
171   L.build_load iPtr id builder
172
173   | A.AMatPlus(id1, idType1, id2, idType2, _) -> let loadedLMatrix1
  ↪ = L.build_load (lookup globalSymbolTable localSymbolTable id1)
  ↪ id1 builder
174   in let loadedLMatrix2 = L.build_load (lookup globalSymbolTable
  ↪ localSymbolTable id2) id2 builder
175   in
176   let _ = L.build_call addMat_func [|loadedLMatrix1;
  ↪ loadedLMatrix2|]
177   "addMat" builder

```

```

178     in
179     loadedLMatrix1
180
181     | A.AMatMinus(id1, idType1, id2, idType2, _) -> let loadedLMatrix1
182     ↪ = L.build_load (lookup globalSymbolTable localSymbolTable id1)
183     ↪ id1 builder
184     in let loadedLMatrix2 = L.build_load (lookup globalSymbolTable
185     ↪ localSymbolTable id2) id2 builder
186     in
187     let _ = L.build_call minusMat_func [|loadedLMatrix1;
188     ↪ loadedLMatrix2|]
189     "minusMat" builder
190     in
191     loadedLMatrix1
192
193     (*| A.Size(id, _, _) ->*)
194     | A.ANoexpr(_) -> L.const_int i32_t 0
195     | A.ABinaryOp(aexpr1, op, aexpr2, _) ->
196         let e1 = genCodeForExpression globalSymbolTable
197         ↪ localSymbolTable memoryMap builder aexpr1 in let e2 =
198         ↪ genCodeForExpression globalSymbolTable
199         ↪ localSymbolTable memoryMap builder aexpr2
200         in
201         (match op with
202         | A.Add      -> L.build_add
203         | A.Sub      -> L.build_sub
204         | A.Mul      -> L.build_mul
205         | A.Div      -> L.build_sdiv
206         | A.And      -> L.build_and
207         | A.Or       -> L.build_or
208         | A.Equal    -> L.build_icmp L.Icmp.Eq
209         | A.Neq      -> L.build_icmp L.Icmp.Ne
210         | A.Less     -> L.build_icmp L.Icmp.Slt
211         | A.Leq      -> L.build_icmp L.Icmp.Sle
212         | A.Greater  -> L.build_icmp L.Icmp.Sgt
213         | A.Geq      -> L.build_icmp L.Icmp.Sge
214         | A.Mod      -> L.build_srem
215         (* TODO: Change the code for Exp
216         * Following code is just a placeholder
217         *)
218         | A.Exp      -> L.build_mul
219         ) e1 e2 "tmp" builder

```

```

215 | A.AUnop(uop, aexpr, exprType) ->
216     let e = genCodeForExpression globalSymbolTable
        ↪ localSymbolTable memoryMap builder aexpr in
217     (match uop with
218     | A.Neg -> L.build_neg
219     | A.Not -> L.build_not) e "tmp" builder
220 | A.ACall ("print", _, [aexpr], _) | A.ACall ("printb", _,
        ↪ [aexpr], _) ->
221 L.build_call printf_func [| int_format_str ;
        ↪ (genCodeForExpression globalSymbolTable localSymbolTable
        ↪ memoryMap builder aexpr)|]
222     "printf" builder
223 | A.ACall(id, _, aExprList, _) -> let actuals = List.map (fun
        ↪ aexpr -> genCodeForExpression globalSymbolTable
        ↪ localSymbolTable memoryMap builder aexpr) aExprList
224     in
225     let (lfunc, afunc) =
        ↪ Hashtbl.find functionTable
        ↪ id
226     in
227     let retType = afunc.A.retType
228     in
229     let result = (match retType
        ↪ with
230     | A.Void ->
        ↪ ""
231     | _ -> id ^
        ↪ "_result")
        ↪ in
232     L.build_call lfunc
        ↪ (Array.of_list actuals)
        ↪ result builder
233 | A.AMatAccess(id, idType, aExprList, nDims, exprType) ->
234     let dimIndices = List.map (fun aexpr ->
        ↪ genCodeForExpression globalSymbolTable
        ↪ localSymbolTable memoryMap builder aexpr) aExprList
235     in
236     let lMatrix = (lookup globalSymbolTable
        ↪ localSymbolTable id)
237     in
238     let loadedLMatrix = L.build_load lMatrix (id ^ "_load")
        ↪ builder
239     in
240     let rec findDimSizes dimSizeList = function

```

```

241         | 0 -> dimSizeList
242         | i -> findDimSizes (
243             (
244                 let dimPtr = (L.build_in_bounds_gep
245                     ↪ loadedLMatrix [|L.const_int i32_t
246                     ↪ i|] (id ^ "_dim_" ^ string_of_int
247                     ↪ i) builder)
248                 in
249                 L.build_load dimPtr (id ^ "_dim_" ^
250                     ↪ string_of_int i ^ "value_") builder
251             )
252             ::dimSizeList) (i - 1)
253
254 in let dimSizes = findDimSizes [] nDims
255 in
256 let (index, _) = List.fold_left2 (fun (result,
257     ↪ productDim) size_i index_i ->
258     let productDimMulIndex = (L.build_mul
259     ↪ productDim index_i ("tmp_") builder)
260     in let result = (L.build_add result
261     ↪ productDimMulIndex ("tmp2_") builder)
262     in
263     let productDim = (L.build_mul productDim size_i
264     ↪ ("tmp3_") builder)
265     in
266     (result, productDim) )
267 ((L.const_int i32_t (1 + nDims)), (L.const_int i32_t 1))
268 (List.rev dimSizes) (List.rev dimIndices)
269 in
270 let elementPtr = L.build_in_bounds_gep loadedLMatrix
271     ↪ [|index|] (id ^ "_element") builder
272 in
273 L.build_load elementPtr (id ^ "_element") builder
274
275 in
276
277 let freeMatrixFun lMatrix id builder =
278     (* FREE: *)
279     let freeMatrix = L.build_load lMatrix (id ^ "_free") builder
280     in
281     let _ = L.build_free freeMatrix builder

```

```

276         in
277         ()
278
279     in
280
281
282     let genCodeForVarDecl globalSymbolTable localSymbolTable memoryMap
283     ↪ builder = function
284     | A.ANodeDecl(_) -> let _ = L.const_int i32_t 0 in ()
285     | A.AMatrix(id, idType, aExprListList, nRows, nCols, _) ->
286         let _ = (if Hashtbl.mem localSymbolTable id then
287             if Hashtbl.mem memoryMap id then
288                 (* free, malloc *)
289                 (* FREE: *)
290                 let _ = freeMatrixFun (lookup
291                 ↪ globalSymbolTable localSymbolTable
292                 ↪ id) id builder
293                 in
294                 (* MALLOC *)
295                 let mallocMatrix = L.build_array_malloc
296                 ↪ i32_t (L.const_int i32_t (nRows *
297                 ↪ nCols + 3)) (id ^ "_malloc") builder
298                 in
299                 let _ = L.build_store mallocMatrix
300                 ↪ (lookup globalSymbolTable
301                 ↪ localSymbolTable id) builder
302                 in ()
303             else
304                 (* malloc, add to memory map *)
305                 (* MALLOC *)
306                 let mallocMatrix = L.build_array_malloc
307                 ↪ i32_t (L.const_int i32_t (nRows *
308                 ↪ nCols + 3)) (id ^ "_malloc") builder
309                 in
310                 let _ = L.build_store mallocMatrix
311                 ↪ (lookup globalSymbolTable
312                 ↪ localSymbolTable id) builder
313                 in
314                 (* Add to memory map *)
315                 Hashtbl.add memoryMap id (lookup
316                 ↪ globalSymbolTable localSymbolTable
317                 ↪ id)
318
319         else if Hashtbl.mem globalSymbolTable id then

```



```

307         (* free, malloc *)
308         (* FREE: *)
309         let _ = freeMatrixFun (lookup
                               ↪ globalSymbolTable localSymbolTable
                               ↪ id) id builder
310         in
311         (* MALLOC *)
312         let mallocMatrix = L.build_array_malloc
                               ↪ i32_t (L.const_int i32_t (nRows *
                               ↪ nCols + 3)) (id ^ "_malloc") builder
313         in
314         let _ = L.build_store mallocMatrix
                               ↪ (lookup globalSymbolTable
                               ↪ localSymbolTable id) builder
315         in
316         ()
317     else
318         (* alloc ptr, malloc, add to local map, add
319         ↪ to memory map *)
320         (* Alloc *)
321         let lMatrix = L.build_alloca mat_t id builder
322         in
323         let _ = Hashtbl.add localSymbolTable id
                               ↪ (lMatrix, idType)
324         in
325         let _ = Hashtbl.add memoryMap id lMatrix
326         in
327         (* MALLOC *)
328         let mallocMatrix = L.build_array_malloc i32_t
                               ↪ (L.const_int i32_t (nRows * nCols + 3))
                               ↪ (id ^ "_malloc") builder
329         in
330         let _ = L.build_store mallocMatrix lMatrix
                               ↪ builder
331         in
332         ()
333     ) in
334     (* Assign dimensions and values *)
335     let lMatrix = (lookup globalSymbolTable
336     ↪ localSymbolTable id)
337     in
338     let loadedLMatrix = L.build_load lMatrix (id ^ "_load")
339     ↪ builder
340     in

```

```

338     let zeroIndexPtr = L.build_in_bounds_gep loadedLMatrix
      ↪ [|L.const_int i32_t 0|] (id ^ "_zero_index")
      ↪ builder
339   in
340   let _ = L.build_store (L.const_int i32_t 2)
      ↪ zeroIndexPtr builder
341   in
342   let oneIndexPtr = L.build_in_bounds_gep loadedLMatrix
      ↪ [|L.const_int i32_t 1|] (id ^ "_one_index") builder
343   in
344   let _ = L.build_store (L.const_int i32_t nRows)
      ↪ oneIndexPtr builder
345   in
346   let secondIndexPtr = L.build_in_bounds_gep
      ↪ loadedLMatrix [|L.const_int i32_t 2|] (id ^
      ↪ "_two_index") builder
347   in
348   let _ = L.build_store (L.const_int i32_t nCols)
      ↪ secondIndexPtr builder
349   in
350     let _ = List.fold_left (fun acc aExprList ->
351 List.fold_left (fun acc aexpr -> let lvalue =
      ↪ genCodeForExpression globalSymbolTable localSymbolTable
      ↪ memoryMap builder aexpr in
352 let ptrIndex = L.build_in_bounds_gep loadedLMatrix
      ↪ [|L.const_int i32_t acc|] (id ^ "_ptr_index") builder in
353 let _ = L.build_store lvalue ptrIndex builder in
      (acc + 1)) acc aExprList
354   ) 3 aExprListList
355   in
356   ()
357
358
359 | A.ADimAssign(id, idType, aExprList, nDims, _) ->
360     let (productSizes, dimSizeList) =
      ↪ List.fold_left (fun (acc, dimSizeLst)
      ↪ aexpr -> let laexpr =
      ↪ genCodeForExpression globalSymbolTable
      ↪ localSymbolTable memoryMap builder
      ↪ aexpr in let dimLst =
      ↪ dimSizeLst @ [laexpr]
361     in
362     (L.build_mul acc laexpr "expr_prod_size"
      ↪ builder, dimLst)) (L.const_int i32_t 1,
363     ↪ []) aExprList

```

```

364 in
365 let matrixSize = (L.build_add productSizes (L.const_int i32_t
  ↪ (nDims + 1)) "mat_size" builder)
366 in
367     let _ = (if Hashtbl.mem localSymbolTable id then
368         if Hashtbl.mem memoryMap id then
369             (* free, malloc *)
370             (* FREE: *)
371             let _ = freeMatrixFun (lookup
  ↪ globalSymbolTable localSymbolTable
  ↪ id) id builder
372 in
373             (* MALLOC *)
374             let mallocMatrix = L.build_array_malloc
  ↪ i32_t matrixSize (id ^ "_malloc")
  ↪ builder
375 in
376             let _ = L.build_store mallocMatrix
  ↪ (lookup globalSymbolTable
  ↪ localSymbolTable id) builder
377 in ()
378 else
379     (* malloc, add to memory map *)
380     (* MALLOC *)
381     let mallocMatrix = L.build_array_malloc
  ↪ i32_t matrixSize (id ^ "_malloc")
  ↪ builder
382 in
383     let _ = L.build_store mallocMatrix
  ↪ (lookup globalSymbolTable
  ↪ localSymbolTable id) builder
384 in
385     (* Add to memory map *)
386     Hashtbl.add memoryMap id (lookup
  ↪ globalSymbolTable localSymbolTable
  ↪ id)
387
388 else if Hashtbl.mem globalSymbolTable id then
389     (* free, malloc *)
390     (* FREE: *)
391     let _ = freeMatrixFun (lookup
  ↪ globalSymbolTable localSymbolTable
  ↪ id) id builder
392 in

```

```

393         (* MALLOC *)
394         let mallocMatrix = L.build_array_malloc
           ↪ i32_t matrixSize (id ^ "_malloc")
           ↪ builder
395         in
396         let _ = L.build_store mallocMatrix
           ↪ (lookup globalSymbolTable
           ↪ localSymbolTable id) builder
397         in
398         ()
399     else
400         (* alloc ptr, malloc, add to local map, add
           ↪ to memory map *)
401         (* Alloc *)
402         let lMatrix = L.build_alloca mat_t id builder
403         in
404         let _ = Hashtbl.add localSymbolTable id
           ↪ (lMatrix, idType)
405         in
406         let _ = Hashtbl.add memoryMap id lMatrix
407         in
408         (* MALLOC *)
409         let mallocMatrix = L.build_array_malloc i32_t
           ↪ matrixSize (id ^ "_malloc") builder
410         in
411         let _ = L.build_store mallocMatrix lMatrix
           ↪ builder
412         in
413         ()
414     ) in
415     let lMatrix = (lookup globalSymbolTable
           ↪ localSymbolTable id)
416     in
417     let loadedLMatrix = L.build_load lMatrix (id ^ "_load")
           ↪ builder
418     in
419     let zeroIndexPtr = L.build_in_bounds_gep loadedLMatrix
           ↪ [|L.const_int i32_t 0|] (id ^ "_zero_index")
           ↪ builder
420     in
421     let _ = L.build_store (matrixSize) zeroIndexPtr builder
422     in
423     let _ = List.fold_left (fun acc lvalue ->

```

```

424     let ptrIndex = L.build_in_bounds_gep loadedLMatrix
      ↪ [|L.const_int i32_t acc|] (id ^ "_ptr_index") builder in
425     let _ = L.build_store lvalue ptrIndex builder in
426             (acc + 1)) 1 dimSizeList
427         in
428     ()
429
430 | A.AMatElementAssign(id, idType, aExprList, aExpr, nDims, _) ->
431     let dimIndices = List.map (fun aexpr ->
      ↪ genCodeForExpression globalSymbolTable
      ↪ localSymbolTable memoryMap builder aexpr) aExprList
432     in
433     let lMatrix = (lookup globalSymbolTable
      ↪ localSymbolTable id)
434     in
435     let loadedLMatrix = L.build_load lMatrix (id ^ "_load")
      ↪ builder
436     in
437     let rec findDimSizes dimSizeList = function
438         | 0 -> dimSizeList
439         | i -> findDimSizes (
440             (
441                 let dimPtr = (L.build_in_bounds_gep
      ↪ loadedLMatrix [|L.const_int i32_t
      ↪ i|] (id ^ "_dim_" ^ string_of_int
      ↪ i) builder)
442                 in
443                 L.build_load dimPtr (id ^ "_dim_" ^
      ↪ string_of_int i ^ "value_") builder
444             )
445             ::dimSizeList) (i - 1)
446
447     in let dimSizes = findDimSizes [] nDims
448     in
449     let (index, _) = List.fold_left2 (fun (result,
450     ↪ productDim) size_i index_i ->
451         let productDimMulIndex = (L.build_mul
      ↪ productDim index_i ("tmp_") builder)
452         in let result = (L.build_add result
      ↪ productDimMulIndex ("tmp2_") builder)
453         in
454         let productDim = (L.build_mul productDim size_i
      ↪ ("tmp3_") builder)

```

```

455         in
456         (result, productDim) )
457     ((L.const_int i32_t (1 + nDims)), (L.const_int i32_t 1))
458     (List.rev dimSizes) (List.rev dimIndices)
459     in
460     let elementPtr = L.build_in_bounds_gep loadedLMatrix
461       ↪ [|index|] (id ^ "_element") builder
462     in
463     let eValue = genCodeForExpression globalSymbolTable
464       ↪ localSymbolTable memoryMap builder aExpr in
465     let _ = L.build_store eValue elementPtr builder in ()
466
467 | A.AExprAssign(id, idType, aExpr, _) ->
468   (
469     match idType with
470     | A.Mat(nDims) ->
471       let loadedLMatrix = genCodeForExpression
472         ↪ globalSymbolTable localSymbolTable memoryMap
473         ↪ builder aExpr
474       in
475       let rec findDimSizes dimSizeList = function
476         | 0 -> dimSizeList
477         | i -> findDimSizes (
478           (
479             let dimPtr = (L.build_in_bounds_gep
480               ↪ loadedLMatrix [|L.const_int i32_t
481               ↪ i|] (id ^ "_dim_" ^ string_of_int
482               ↪ i) builder)
483             in
484             L.build_load dimPtr (id ^ "_dim_" ^
485               ↪ string_of_int i ^ "value_") builder
486           )
487           ::dimSizeList) (i - 1)
488       in let dimSizes = findDimSizes [] nDims
489       in
490         let productSizes = List.fold_left (fun acc
491           ↪ laexpr ->
492           (L.build_mul acc laexpr "expr_prod_size"
493           ↪ builder)) (L.const_int i32_t 1) dimSizes
494         in

```

```

488         let matrixSize = (L.build_add productSizes
489           ↪ (L.const_int i32_t (nDims + 1)) "mat_size"
490           ↪ builder)
491
492     in
493
494     let _ = (if Hashtbl.mem localSymbolTable id then
495       if Hashtbl.mem memoryMap id then
496         (* free, malloc *)
497         (* FREE: *)
498         let _ = freeMatrixFun (lookup
499           ↪ globalSymbolTable localSymbolTable
500           ↪ id) id builder
501         in
502         (* MALLOC *)
503         let mallocMatrix = L.build_array_malloc
504           ↪ i32_t matrixSize (id ^ "_malloc")
505           ↪ builder
506         in
507         let _ = L.build_store mallocMatrix
508           ↪ (lookup globalSymbolTable
509           ↪ localSymbolTable id) builder
510         in ()
511       else
512         (* malloc, add to memory map *)
513         (* MALLOC *)
514         let mallocMatrix = L.build_array_malloc
515           ↪ i32_t matrixSize (id ^ "_malloc")
516           ↪ builder
517         in
518         let _ = L.build_store mallocMatrix
519           ↪ (lookup globalSymbolTable
520           ↪ localSymbolTable id) builder
521         in
522         (* Add to memory map *)
523         Hashtbl.add memoryMap id (lookup
524           ↪ globalSymbolTable localSymbolTable
525           ↪ id)
526
527     else if Hashtbl.mem globalSymbolTable id then
528       (* free, malloc *)
529       (* FREE: *)
530       let _ = freeMatrixFun (lookup
531         ↪ globalSymbolTable localSymbolTable
532         ↪ id) id builder

```

```

516         in
517         (* MALLOC *)
518         let mallocMatrix = L.build_array_malloc
           ↪ i32_t matrixSize (id ^ "_malloc")
           ↪ builder
519         in
520         let _ = L.build_store mallocMatrix
           ↪ (lookup globalSymbolTable
           ↪ localSymbolTable id) builder
521         in
522         ()
523     else
524         (* alloc ptr, malloc, add to local map, add
           ↪ to memory map *)
525         (* Alloc *)
526         let lMatrix = L.build_alloca mat_t id builder
527         in
528         let _ = Hashtbl.add localSymbolTable id
           ↪ (lMatrix, idType)
529         in
530         let _ = Hashtbl.add memoryMap id lMatrix
531         in
532         (* MALLOC *)
533         let mallocMatrix = L.build_array_malloc i32_t
           ↪ matrixSize (id ^ "_malloc") builder
534         in
535         let _ = L.build_store mallocMatrix lMatrix
           ↪ builder
536         in
537         ()
538     ) in
539     let lMatrix = (lookup globalSymbolTable
           ↪ localSymbolTable id)
540     in
541     let loadedAllocMat = L.build_load lMatrix (id ^
           ↪ "_load") builder
542     in
543     let _ = L.build_call copyMat_func [|loadedLMatrix;
           ↪ loadedAllocMat|]
544     "copyMat" builder
545     in
546     ()

```



```

547 | typ -> if (Hashtbl.mem localSymbolTable id) ||
    ↪ (Hashtbl.mem globalSymbolTable id) then let lid =
    ↪ (lookup globalSymbolTable localSymbolTable id) in let _
    ↪ = (L.build_store (genCodeForExpression
    ↪ globalSymbolTable localSymbolTable memoryMap builder
    ↪ aExpr) lid builder) in ()
548     else
549         let local = L.build_alloca (ltypeOfType typ) id
    ↪ builder
550         in
551         ignore (L.build_store (genCodeForExpression
    ↪ globalSymbolTable localSymbolTable memoryMap
    ↪ builder aExpr) local builder);
552         Hashtbl.add localSymbolTable id (local, typ)
553     )
554
555 in
556
557 let addTerminal builder f =
558     (match L.block_terminator (L.insertion_block builder) with
559     Some _ -> ()
560     | None -> ignore (f builder))
561 in
562
563 let rec genCodeForStatements ?contBB:(contBB = None)
    ↪ globalSymbolTable localSymbolTable memoryMap builder = function
564 | A.ABlock(astatementList, _) -> let newGlobalSymbolTable =
    ↪ mergeSymbolTables globalSymbolTable localSymbolTable in
565     let newLocalSymbolTable = Hashtbl.create
    ↪ 100 in
566     let newMemoryMap = Hashtbl.create 100 in
567     List.fold_left (fun builder astatement
    ↪ -> genCodeForStatements
    ↪ ~contBB:contBB newGlobalSymbolTable
    ↪ newLocalSymbolTable newMemoryMap
    ↪ builder astatement) builder
    ↪ astatementList
568 | A.AExpr(aExpr, _) -> let _ = genCodeForExpression
    ↪ globalSymbolTable localSymbolTable memoryMap builder aExpr
    ↪ in builder
569
570 | A.AVarDecl(aVarDecl, _) -> let _ = genCodeForVarDecl
    ↪ globalSymbolTable localSymbolTable memoryMap builder
    ↪ aVarDecl

```

```

571         in builder
572
573     | A.AReturn(retType, aExpr, _) -> ignore (match afunc.A.retType
574         ↪ with
575
576             | A.Void ->
577                 ↪ L.build_ret_void
578                 ↪ builder
579             | _ -> L.build_ret
580                 ↪ (genCodeForExpression
581                     ↪ globalSymbolTable
582                     ↪ localSymbolTable
583                     ↪ memoryMap builder
584                     ↪ aExpr)
585
586                                     ↪ builder);
587                                     ↪ builder
588
589 | A.AIf (predicate, thenStatement, elseStatement, _) ->
590     let boolVal = genCodeForExpression globalSymbolTable
591         ↪ localSymbolTable memoryMap builder predicate in
592     let mergeBB = L.append_block context "merge"
593         ↪ theFunction in
594
595     let thenBb = L.append_block context "then" theFunction
596         ↪ in
597     addTerminal (genCodeForStatements ~contBB:contBB
598         ↪ globalSymbolTable localSymbolTable memoryMap
599         ↪ (L.builder_at_end context thenBb) thenStatement)
600         (L.build_br mergeBB);
601
602     let elseBb = L.append_block context "else" theFunction
603         ↪ in
604     addTerminal (genCodeForStatements ~contBB:contBB
605         ↪ globalSymbolTable localSymbolTable memoryMap
606         ↪ (L.builder_at_end context elseBb) elseStatement)
607         (L.build_br mergeBB);
608
609     ignore (L.build_cond_br boolVal thenBb elseBb
610         ↪ builder);
611     L.builder_at_end context mergeBB
612
613 | A.AWhile (predicate, body, _) ->
614     let predBB = L.append_block context "while"
615         ↪ theFunction in
616     ignore (L.build_br predBB builder);

```

```

595
596     let bodyBB = L.append_block context "while_body"
      ↪ theFunction in
597     addTerminal (genCodeForStatements globalSymbolTable
      ↪ localSymbolTable memoryMap ~contBB:(Some predBB)
      ↪ (L.builder_at_end context bodyBB) body)
      (L.build_br predBB);
598
599
600     let predBuilder = L.builder_at_end context predBB in
601     let boolVal = genCodeForExpression globalSymbolTable
      ↪ localSymbolTable memoryMap predBuilder predicate
      ↪ in
602
603     let mergeBB = L.append_block context "merge"
      ↪ theFunction in
604     ignore (L.build_cond_br boolVal bodyBB mergeBB
      ↪ predBuilder);
605     L.builder_at_end context mergeBB
606
607
608
609 | A.AFor(aVarDecl1, aExpr, aVarDecl2, aStatement, _) ->
610     (let newGlobalSymbolTable = mergeSymbolTables
      ↪ globalSymbolTable localSymbolTable in
611     let newLocalSymbolTable = Hashtbl.create 100 in
612     let newMemoryMap = Hashtbl.create 100 in
613     let builder = genCodeForStatements ~contBB:contBB
      ↪ newGlobalSymbolTable newLocalSymbolTable
      ↪ newMemoryMap builder (A.AVarDecl(aVarDecl1,
      ↪ A.Void))
614     in
615     let incrBB = L.append_block context "forincr"
      ↪ theFunction in
616     let _ = (genCodeForStatements ~contBB:contBB
      ↪ newGlobalSymbolTable newLocalSymbolTable
      ↪ newMemoryMap (L.builder_at_end context incrBB)
      ↪ (A.AVarDecl(aVarDecl2, A.Void)))
617     in
618     let forBB = L.append_block context "for" theFunction in
619     let _ = L.build_br forBB builder in
620     let _ = L.build_br forBB (L.builder_at_end context
      ↪ incrBB) in
621     let predBuilder = L.builder_at_end context forBB in

```

```

622     let boolVal = genCodeForExpression newGlobalSymbolTable
        ↪ newLocalSymbolTable newMemoryMap predBuilder aExpr
        ↪ in
623     let bodyBB = L.append_block context "forbody"
        ↪ theFunction in
624     let _ = addTerminal (genCodeForStatements
        ↪ newGlobalSymbolTable newLocalSymbolTable
        ↪ newMemoryMap ~contBB:(Some incrBB)
        ↪ (L.builder_at_end context bodyBB) aStatement)
        ↪ (L.build_br incrBB)
625     in
626     let mergeBB = L.append_block context "merge"
        ↪ theFunction in
627     ignore (L.build_cond_br boolVal bodyBB mergeBB
        ↪ predBuilder);
628     L.builder_at_end context mergeBB)
629
630
631 | A.AExit(_) -> builder
632 | A.ABreak(_) -> builder
633 | A.AForEachLoop (_, _, _, _, _, _, _) -> builder
634 | A.AContinue(_) -> let _ = (match contBB with
635     | Some x -> L.build_br x builder
636     | _ -> printError ("Code should never reach
        ↪ here!")) in builder
637
638     in
639     let builder = genCodeForStatements globalSymbolTable
        ↪ localSymbolTable memoryMap builder (A.ABlock
        ↪ (afunc.A.abody, A.Void)) in
640         addTerminal builder (match afunc.A.retType with
641             | A.Void -> L.build_ret_void
642             | t -> L.build_ret (initOfType
        ↪ t))
643
644     in
645     (* Generate code for function definitions *)
646     let _ = List.iter generateFunctionBody functions
647     in
648
649 the_module

```

## 8.7 matcv.ml

```
1  (* Top-level of the MicroC compiler: scan & parse the input,
2     check the resulting AST, generate LLVM IR, and dump the module *)
3  open Ast
4  open Lllvm
5
6  type action = Ast | LLVM_IR | Compile
7
8  let _ =
9     let action = if Array.length Sys.argv > 1 then
10        List.assoc Sys.argv.(1) [ ("-a", Ast);           (* Print the AST only
11        ↪ *)
12        ("l", LLVM_IR);  (* Generate LLVM, don't
13        ↪ check *)
14        ("-c", Compile) ] (* Generate, check LLVM
15        ↪ IR *)
16     else Compile in
17     let lexbuf = Lexing.from_channel stdin in
18     let ast = Parser.program Scanner.token lexbuf in
19     let sast = Semant.check_semantics ast in
20     match action with
21     | Ast ->
22       print_string(Ast.string_of_program(sast))
23     | LLVM_IR -> print_string (Lllvm.string_of_llmodule (Codegen.translate
24     ↪ sast))
25     | Compile -> let m = Codegen.translate sast in
26       Lllvm_analysis.assert_valid_module m;
27       print_string (Lllvm.string_of_llmodule m)
```

## 8.8 Makefile

```
1  # Make sure ocamlbuild can find opam-managed packages: first run
2  #
3  # eval `opam config env`
4
5  # Easiest way to build: using ocamlbuild, which in turn uses ocamlfind
6
7  .PHONY : matcv.native
8
9  matcv.native :
10     ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis \
11     matcv.native
12
```

```

13 # "make clean" removes all generated files
14
15 .PHONY : clean
16 clean :
17     ocamlbuild -clean
18     rm -rf testall.log *.diff matcv scanner.ml parser.ml parser.mli
19     rm -rf *.cmx *.cmi *.cmo *.cmx *.o
20
21 # More detailed: build using ocamlc/ocamlopt + ocamlfind to locate
22   ↪ LLVM
23
24 OBJS = ast.cmx codegen.cmx parser.cmx scanner.cmx semant.cmx matcv.cmx
25
26 matcv : $(OBJS)
27     ocamlfind ocamlopt -linkpkg -package llvm -package
28     ↪ llvm.analysis $(OBJS) -o matcv
29
30 scanner.ml : scanner.mll
31     ocamllex scanner.mll
32
33 parser.ml parser.mli : parser.mly
34     ocamlyacc parser.mly
35
36 %.cmo : %.ml
37     ocamlc -c $<
38
39 %.cmi : %.mli
40     ocamlc -c $<
41
42 %.cmx : %.ml
43     ocamlfind ocamlopt -c -package llvm $<
44
45 ### Generated by "ocamldep *.ml *.mli" after building scanner.ml and
46   ↪ parser.ml
47
48 ast.cmo :
49 ast.cmx :
50 codegen.cmo : ast.cmo
51 codegen.cmx : ast.cmx
52 matcv.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo
53 matcv.cmx : semant.cmx scanner.cmx parser.cmx codegen.cmx ast.cmx
54 parser.cmo : ast.cmo parser.cmi
55 parser.cmx : ast.cmx parser.cmi
56 scanner.cmo : parser.cmi

```

```

53 scanner.cmx : parser.cmx
54 semant.cmo : ast.cmo
55 semant.cmx : ast.cmx
56 parser.cmi : ast.cmo
57
58 # Building the tarball
59
60 TESTS = add1 arith1 arith2 arith3 fib for1 for2 func1 func2
61 ↪ func3          \
62   func4 func5 func6 func7 func8 gcd2 gcd global1 global2
63   ↪ global3          \
64   hello if1 if2 if3 if4 if5 local1 local2 ops1 ops2 var1
65   ↪ var2              \
66   while1 while2
67
68 FAILS = assign1 assign2 assign3 dead1 dead2 expr1 expr2 for1
69 ↪ for2          \
70   for3 for4 for5 func1 func2 func3 func4 func5 func6 func7
71   ↪ func8          \
72   func9 global1 global2 if1 if2 if3 nomain return1 return2
73   ↪ while1          \
74   while2
75
76 TESTFILES = $(TESTS:%=test-%.mc) $(TESTS:%=test-%.out) \
77             $(FAILS:%=fail-%.mc) $(FAILS:%=fail-%.err)
78
79 TARFILES = ast.ml codegen.ml Makefile matcv.ml parser.mly README
80 ↪ scanner.mll \
81   semant.ml testall.sh $(TESTFILES:%=tests/%)
82
83 matcv-llvm.tar.gz : $(TARFILES)
84   cd .. && tar czf matcv-llvm/matcv-llvm.tar.gz \
85   $(TARFILES:%=matcv-llvm/%)

```

## 9. References

1. <https://llvm.moe/ocaml-3.7/Llvm.html>
2. <https://www.wzdftpd.net/blog/ocaml-llvm-01.html>
3. <https://www.cs.cornell.edu/courses/cs3110/2011sp/lectures/lec26-type-inference/type-inference.htm>