MatCV - Project Report

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

Contents

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:

```
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:

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

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

```
function addElements(matrix1, matrix2){

        sum = matrix1 +. matrix2;

        return sum;
}
```

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.

```
a = 3; /* Type of a inferred as Int */
b = [3][4]; /* Type of b inferred as Mat(2)*/
/*which means that b is a matrix of 2 dimensions */
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 expresions 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:

```
a = 2;
b = 3;
a + b;
```

```
a*b;
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.

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

```
if (d == 3){
        d = d + 1;
}
else{
        d = d - 1;
} /* else is optional in the conditional check */

if (d == 3)
        d = d + 1;
/* braces are also optional*/

for (i = 1; i < 10; i = i+1 ){
        d = d + 1;
} /* For loop - all expressions are optional in the loop initialization */


for (; d < 10;){
        d = d + 1;
    if (d == 6){
            continue; /* goes to beginning of of the loop */
    }
}

function test(){
        a = 2;
    return a; /* returns control back to the calling function */
}
```

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:

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

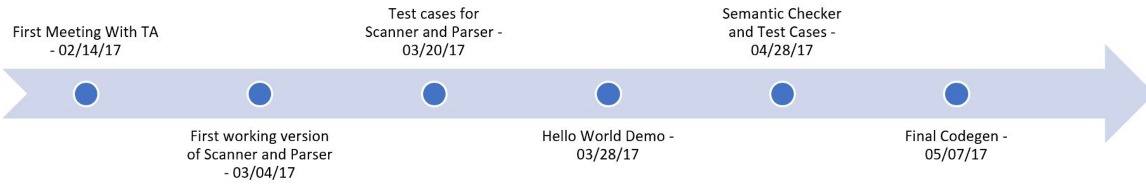

Figure 1: Timeline of project

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

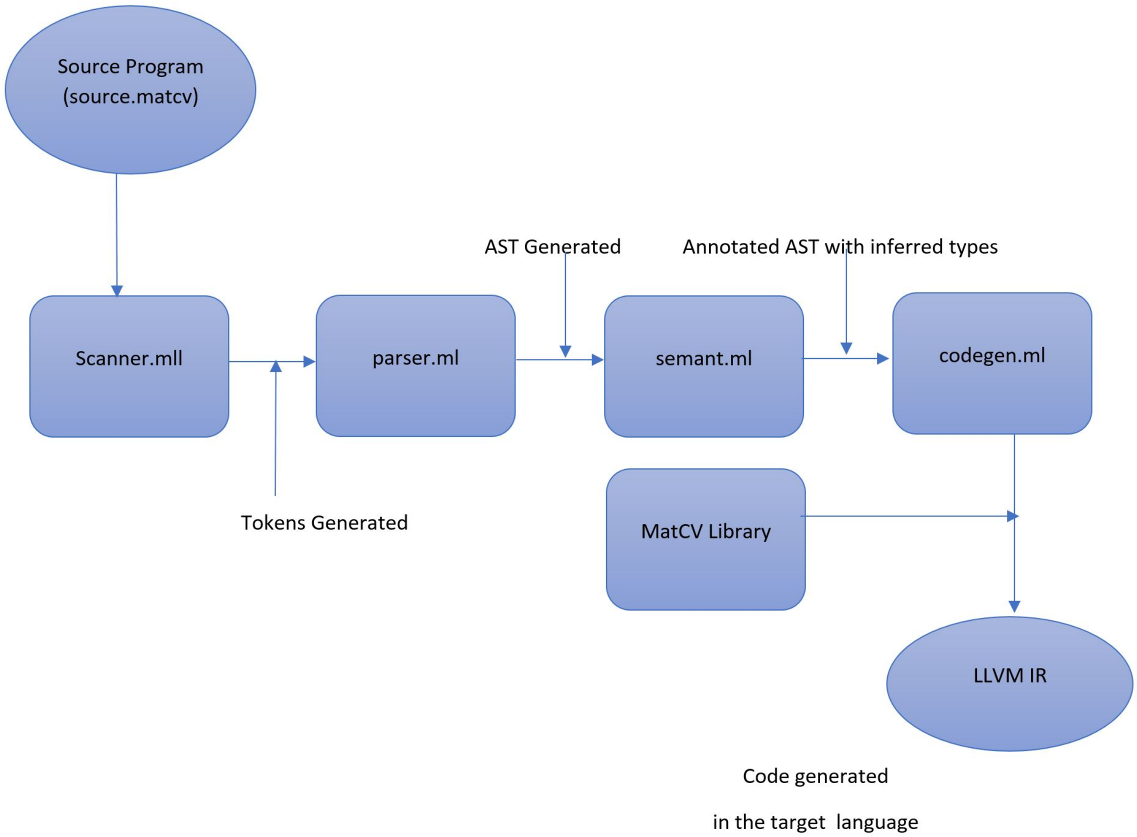

Figure 2: Architecture diagram for the compiler

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:

```bash
#!/bin/bash

TEST_FILES=`find ./Tests -type f -not -name "*.out"`


```

```
echo "Test $I)"
if [ ! -f matcv.native ]
then
    make
fi

I=0
for FILE in $TEST_FILES
do
    echo "$I) Running test case for $FILE:"

    cat library.matcv $FILE > __run__
    ./matcv.native < __run__ &> __out__
    if [ $? != 0 ]
    then
        echo "Test failed for1: $FILE"
    else
        lli __out__ &>> __output__
        if [ $? != 0 ]
        then
            echo "Test failed for2: $FILE"
        else
            diff __output__ "$FILE".out
            if [ $? != 0 ]
            then
                echo "Test failed for2: $FILE"
            else
                echo "Test passed for: $FILE"
            fi
        fi
    fi

    I=`expr $I + 1`


    rm -f __run__ __out__ __output__

    echo "----------------------------"


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

```
A = [3];
a[0] = 1;
a[1] = 2;
a[2] = 3;

n = 4;
m =  3;

result = coinChange(a,m,n);
print(result);

function coinChange(S, m, n){
    table = [n+1][m];
    x = 0;
    i = 0;
    j = 0;
    y = 0;
    for (i = 0; i < m; i=i+1)
    {
        table[0][i] = 1;
    }

    for (i = 1; i < n+1; i=i+1)
    {
        for (j = 0; j < m; j=j+1)
        {
            if (i - S[j] >= 0){
                temp = i - S[j];
                x = table[temp][j];
            }
            else{
                x = 0;
            }
            if (j >= 1){

                y = table[i][j-1];
```

```
            }
            else{
                y = 0;
            }
            table[i][j] = x + y;
        }
    }
    ans = table[n][m-1];
    return ans;
```

Target Language - LLVM IR

```
; ModuleID = 'MatCV'

@a = global i32* null
@n = global i32 0
@m = global i32 0
@result = global i32 0
@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt1 = private unnamed_addr constant [4 x i8] c"%d\0A\00"

define i32 @main() {
entry:
  %a_free = load i32** @a
  %0 = bitcast i32* %a_free to i8*
  tail call void @free(i8* %0)
  %malloccall = tail call i8* @malloc(i32 mul (i32 ptrtoint (i32*
   ↪  getelementptr (i32* null, i32 1) to i32), i32 5))
  %a_malloc = bitcast i8* %malloccall to i32*
  store i32* %a_malloc, i32** @a
  %a_load = load i32** @a
  %a_zero_index = getelementptr inbounds i32* %a_load, i32 0
  store i32 5, i32* %a_zero_index
  %a_ptr_index = getelementptr inbounds i32* %a_load, i32 1
  store i32 3, i32* %a_ptr_index
  %a_load1 = load i32** @a
  %a_dim_1 = getelementptr inbounds i32* %a_load1, i32 1
  %a_dim_1value_ = load i32* %a_dim_1
  %tmp3_ = mul i32 1, %a_dim_1value_
  %a_element = getelementptr inbounds i32* %a_load1, i32 2
  store i32 1, i32* %a_element
  %a_load2 = load i32** @a
  %a_dim_13 = getelementptr inbounds i32* %a_load2, i32 1
  %a_dim_1value_4 = load i32* %a_dim_13
```

```
  %tmp3_5 = mul i32 1, %a_dim_1value_4
  %a_element6 = getelementptr inbounds i32* %a_load2, i32 3
  store i32 2, i32* %a_element6
  %a_load7 = load i32** @a
  %a_dim_18 = getelementptr inbounds i32* %a_load7, i32 1
  %a_dim_1value_9 = load i32* %a_dim_18
  %tmp3_10 = mul i32 1, %a_dim_1value_9
  %a_element11 = getelementptr inbounds i32* %a_load7, i32 4
  store i32 3, i32* %a_element11
  store i32 4, i32* @n
  store i32 3, i32* @m
  %a = load i32** @a
  %m = load i32* @m
  %n = load i32* @n
  %coinChange_result = call i32 @coinChange(i32* %a, i32 %m, i32 %n)
  store i32 %coinChange_result, i32* @result
  %result = load i32* @result
  %printf = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4
   ↪  x i8]* @fmt, i32 0, i32 0), i32 %result)
  ret i32 0
}

define i32 @coinChange(i32* %S, i32 %m, i32 %n) {
entry:
  %S1 = alloca i32*
  store i32* %S, i32** %S1
  %m2 = alloca i32
  store i32 %m, i32* %m2
  %n3 = alloca i32
  store i32 %n, i32* %n3
  %n4 = load i32* %n3
  %tmp = add i32 %n4, 1
  %expr_prod_size = mul i32 1, %tmp
  %m5 = load i32* %m2
  %expr_prod_size6 = mul i32 %expr_prod_size, %m5
  %mat_size = add i32 %expr_prod_size6, 3
  %table = alloca i32*
  %mallocsize = mul i32 %mat_size, ptrtoint (i32* getelementptr (i32*
   ↪  null, i32 1) to i32)
  %malloccall = tail call i8* @malloc(i32 %mallocsize)
  %table_malloc = bitcast i8* %malloccall to i32*
  store i32* %table_malloc, i32** %table
  %table_load = load i32** %table
```

```
  %table_zero_index = getelementptr inbounds i32* %table_load, i32 0
  store i32 %mat_size, i32* %table_zero_index
  %table_ptr_index = getelementptr inbounds i32* %table_load, i32 1
  store i32 %tmp, i32* %table_ptr_index
  %table_ptr_index7 = getelementptr inbounds i32* %table_load, i32 2
  store i32 %m5, i32* %table_ptr_index7
  %x = alloca i32
  store i32 0, i32* %x
  %i = alloca i32
  store i32 0, i32* %i
  %j = alloca i32
  store i32 0, i32* %j
  %y = alloca i32
  store i32 0, i32* %y
  store i32 0, i32* %i
  br label %for

forincr:                                          ; preds = %forbody
  %i8 = load i32* %i
  %tmp9 = add i32 %i8, 1
  store i32 %tmp9, i32* %i
  br label %for

for:                                              ; preds = %forincr,
 ↪  %entry
  %i10 = load i32* %i
  %m11 = load i32* %m2
  %tmp12 = icmp slt i32 %i10, %m11
  br i1 %tmp12, label %forbody, label %merge

forbody:                                          ; preds = %for
  %i13 = load i32* %i
  %table_load14 = load i32** %table
  %table_dim_2 = getelementptr inbounds i32* %table_load14, i32 2
  %table_dim_2value_ = load i32* %table_dim_2
  %table_dim_1 = getelementptr inbounds i32* %table_load14, i32 1
  %table_dim_1value_ = load i32* %table_dim_1
  %tmp_ = mul i32 1, %i13
  %tmp2_ = add i32 3, %tmp_
  %tmp3_ = mul i32 1, %table_dim_2value_
  %tmp_15 = mul i32 %tmp3_, 0
  %tmp2_16 = add i32 %tmp2_, %tmp_15
  %tmp3_17 = mul i32 %tmp3_, %table_dim_1value_
```

```
  %table_element = getelementptr inbounds i32* %table_load14, i32 %tmp2_16
  store i32 1, i32* %table_element
  br label %forincr

merge:                                            ; preds = %for
  store i32 1, i32* %i
  br label %for21

forincr18:                                        ; preds = %merge108
  %i19 = load i32* %i
  %tmp20 = add i32 %i19, 1
  store i32 %tmp20, i32* %i
  br label %for21

for21:                                            ; preds = %forincr18, %merge
  %i22 = load i32* %i
  %n23 = load i32* %n3
  %tmp24 = add i32 %n23, 1
  %tmp25 = icmp slt i32 %i22, %tmp24
  br i1 %tmp25, label %forbody26, label %merge109

forbody26:                                        ; preds = %for21
  store i32 0, i32* %j
  br label %for30

forincr27:                                        ; preds = %merge72
  %j28 = load i32* %j
  %tmp29 = add i32 %j28, 1
  store i32 %tmp29, i32* %j
  br label %for30

for30:                                            ; preds = %forincr27, %forbody26
  %j31 = load i32* %j
  %m32 = load i32* %m2
  %tmp33 = icmp slt i32 %j31, %m32
  br i1 %tmp33, label %forbody34, label %merge108

forbody34:                                        ; preds = %for30
  %i35 = load i32* %i
  %j36 = load i32* %j
  %S_load = load i32** %S1
```

```
  %S_dim_1 = getelementptr inbounds i32* %S_load, i32 1
  %S_dim_1value_ = load i32* %S_dim_1
  %tmp_37 = mul i32 1, %j36
  %tmp2_38 = add i32 2, %tmp_37
  %tmp3_39 = mul i32 1, %S_dim_1value_
  %S_element = getelementptr inbounds i32* %S_load, i32 %tmp2_38
  %S_element40 = load i32* %S_element
  %tmp41 = sub i32 %i35, %S_element40
  %tmp42 = icmp sge i32 %tmp41, 0
  br i1 %tmp42, label %then, label %else

merge43:                                          ; preds = %else, %then
  %j70 = load i32* %j
  %tmp71 = icmp sge i32 %j70, 1
  br i1 %tmp71, label %then73, label %else90

then:                                             ; preds = %forbody34
  %temp = alloca i32
  %i44 = load i32* %i
  %j45 = load i32* %j
  %S_load46 = load i32** %S1
  %S_dim_147 = getelementptr inbounds i32* %S_load46, i32 1
  %S_dim_1value_48 = load i32* %S_dim_147
  %tmp_49 = mul i32 1, %j45
  %tmp2_50 = add i32 2, %tmp_49
  %tmp3_51 = mul i32 1, %S_dim_1value_48
  %S_element52 = getelementptr inbounds i32* %S_load46, i32 %tmp2_50
  %S_element53 = load i32* %S_element52
  %tmp54 = sub i32 %i44, %S_element53
  store i32 %tmp54, i32* %temp
  %temp55 = load i32* %temp
  %j56 = load i32* %j
  %table_load57 = load i32** %table
  %table_dim_258 = getelementptr inbounds i32* %table_load57, i32 2
  %table_dim_2value_59 = load i32* %table_dim_258
  %table_dim_160 = getelementptr inbounds i32* %table_load57, i32 1
  %table_dim_1value_61 = load i32* %table_dim_160
  %tmp_62 = mul i32 1, %j56
  %tmp2_63 = add i32 3, %tmp_62
  %tmp3_64 = mul i32 1, %table_dim_2value_59
  %tmp_65 = mul i32 %tmp3_64, %temp55
  %tmp2_66 = add i32 %tmp2_63, %tmp_65
  %tmp3_67 = mul i32 %tmp3_64, %table_dim_1value_61
```

```
  %table_element68 = getelementptr inbounds i32* %table_load57, i32 %tmp2_66
  %table_element69 = load i32* %table_element68
  store i32 %table_element69, i32* %x
  br label %merge43

else:                                             ; preds = %forbody34
  store i32 0, i32* %x
  br label %merge43

merge72:                                          ; preds = %else90, %then73
  %i91 = load i32* %i
  %j92 = load i32* %j
  %table_load93 = load i32** %table
  %table_dim_294 = getelementptr inbounds i32* %table_load93, i32 2
  %table_dim_2value_95 = load i32* %table_dim_294
  %table_dim_196 = getelementptr inbounds i32* %table_load93, i32 1
  %table_dim_1value_97 = load i32* %table_dim_196
  %tmp_98 = mul i32 1, %j92
  %tmp2_99 = add i32 3, %tmp_98
  %tmp3_100 = mul i32 1, %table_dim_2value_95
  %tmp_101 = mul i32 %tmp3_100, %i91
  %tmp2_102 = add i32 %tmp2_99, %tmp_101
  %tmp3_103 = mul i32 %tmp3_100, %table_dim_1value_97
  %table_element104 = getelementptr inbounds i32* %table_load93, i32 %tmp2_102
  %x105 = load i32* %x
  %y106 = load i32* %y
  %tmp107 = add i32 %x105, %y106
  store i32 %tmp107, i32* %table_element104
  br label %forincr27

then73:                                           ; preds = %merge43
  %i74 = load i32* %i
  %j75 = load i32* %j
  %tmp76 = sub i32 %j75, 1
  %table_load77 = load i32** %table
  %table_dim_278 = getelementptr inbounds i32* %table_load77, i32 2
  %table_dim_2value_79 = load i32* %table_dim_278
  %table_dim_180 = getelementptr inbounds i32* %table_load77, i32 1
  %table_dim_1value_81 = load i32* %table_dim_180
  %tmp_82 = mul i32 1, %tmp76
  %tmp2_83 = add i32 3, %tmp_82
```

```
  %tmp3_84 = mul i32 1, %table_dim_2value_79
  %tmp_85 = mul i32 %tmp3_84, %i74
  %tmp2_86 = add i32 %tmp2_83, %tmp_85
  %tmp3_87 = mul i32 %tmp3_84, %table_dim_1value_81
  %table_element88 = getelementptr inbounds i32* %table_load77, i32 %tmp2_86
  %table_element89 = load i32* %table_element88
  store i32 %table_element89, i32* %y
  br label %merge72

else90:                                           ; preds = %merge43
  store i32 0, i32* %y
  br label %merge72

merge108:                                         ; preds = %for30
  br label %forincr18

merge109:                                         ; preds = %for21
  %ans = alloca i32
  %n110 = load i32* %n3
  %m111 = load i32* %m2
  %tmp112 = sub i32 %m111, 1
  %table_load113 = load i32** %table
  %table_dim_2114 = getelementptr inbounds i32* %table_load113, i32 2
  %table_dim_2value_115 = load i32* %table_dim_2114
  %table_dim_1116 = getelementptr inbounds i32* %table_load113, i32 1
  %table_dim_1value_117 = load i32* %table_dim_1116
  %tmp_118 = mul i32 1, %tmp112
  %tmp2_119 = add i32 3, %tmp_118
  %tmp3_120 = mul i32 1, %table_dim_2value_115
  %tmp_121 = mul i32 %tmp3_120, %n110
  %tmp2_122 = add i32 %tmp2_119, %tmp_121
  %tmp3_123 = mul i32 %tmp3_120, %table_dim_1value_117
  %table_element124 = getelementptr inbounds i32* %table_load113, i32 %tmp2_122
  %table_element125 = load i32* %table_element124
  store i32 %table_element125, i32* %ans
  %ans126 = load i32* %ans
  ret i32 %ans126
}

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

declare i32 @copyMat(i32*, i32*)
```

```
declare i32 @minusMat(i32*, i32*)

declare i32 @addMat(i32*, i32*)

declare void @free(i8*)

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

6.5 Program 2

This code snippet illustrates the core functionality f 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

```
a = {1,2,3;4,5,6;7,8,9};
b = {9,8,7;6,5,4;3,2,1};

a +. b;
printMatrix(a);
a -. b;
printMatrix(a);


function printMatrix(mat)
{
    print (mat[0][0]);
    return 0;
}
```

Target Language - LLVM IR

```
; ModuleID = 'MatCV'

@a = global i32* null
@b = global i32* null
@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@fmt1 = private unnamed_addr constant [4 x i8] c"%d\0A\00"

define i32 @main() {
entry:
  %a_free = load i32** @a
  %0 = bitcast i32* %a_free to i8*
```

```
tail call void @free(i8* %0)
%malloccall = tail call i8* @malloc(i32 mul (i32 ptrtoint (i32* getelementptr (i32* null, i32 1) to i32), i32 12))
%a_malloc = bitcast i8* %malloccall to i32*
store i32* %a_malloc, i32** @a
%a_load = load i32** @a
%a_zero_index = getelementptr inbounds i32* %a_load, i32 0
store i32 2, i32* %a_zero_index
%a_one_index = getelementptr inbounds i32* %a_load, i32 1
store i32 3, i32* %a_one_index
%a_two_index = getelementptr inbounds i32* %a_load, i32 2
store i32 3, i32* %a_two_index
%a_ptr_index = getelementptr inbounds i32* %a_load, i32 3
store i32 1, i32* %a_ptr_index
%a_ptr_index1 = getelementptr inbounds i32* %a_load, i32 4
store i32 2, i32* %a_ptr_index1
%a_ptr_index2 = getelementptr inbounds i32* %a_load, i32 5
store i32 3, i32* %a_ptr_index2
%a_ptr_index3 = getelementptr inbounds i32* %a_load, i32 6
store i32 4, i32* %a_ptr_index3
%a_ptr_index4 = getelementptr inbounds i32* %a_load, i32 7
store i32 5, i32* %a_ptr_index4
%a_ptr_index5 = getelementptr inbounds i32* %a_load, i32 8
store i32 6, i32* %a_ptr_index5
%a_ptr_index6 = getelementptr inbounds i32* %a_load, i32 9
store i32 7, i32* %a_ptr_index6
%a_ptr_index7 = getelementptr inbounds i32* %a_load, i32 10
store i32 8, i32* %a_ptr_index7
%a_ptr_index8 = getelementptr inbounds i32* %a_load, i32 11
store i32 9, i32* %a_ptr_index8
%b_free = load i32** @b
%1 = bitcast i32* %b_free to i8*
tail call void @free(i8* %1)
%malloccall9 = tail call i8* @malloc(i32 mul (i32 ptrtoint (i32* getelementptr (i32* null, i32 1) to i32), i32 12))
%b_malloc = bitcast i8* %malloccall9 to i32*
store i32* %b_malloc, i32** @b
%b_load = load i32** @b
%b_zero_index = getelementptr inbounds i32* %b_load, i32 0
store i32 2, i32* %b_zero_index
%b_one_index = getelementptr inbounds i32* %b_load, i32 1
store i32 3, i32* %b_one_index
%b_two_index = getelementptr inbounds i32* %b_load, i32 2
```

```
  store i32 3, i32* %b_two_index
  %b_ptr_index = getelementptr inbounds i32* %b_load, i32 3
  store i32 9, i32* %b_ptr_index
  %b_ptr_index10 = getelementptr inbounds i32* %b_load, i32 4
  store i32 8, i32* %b_ptr_index10
  %b_ptr_index11 = getelementptr inbounds i32* %b_load, i32 5
  store i32 7, i32* %b_ptr_index11
  %b_ptr_index12 = getelementptr inbounds i32* %b_load, i32 6
  store i32 6, i32* %b_ptr_index12
  %b_ptr_index13 = getelementptr inbounds i32* %b_load, i32 7
  store i32 5, i32* %b_ptr_index13
  %b_ptr_index14 = getelementptr inbounds i32* %b_load, i32 8
  store i32 4, i32* %b_ptr_index14
  %b_ptr_index15 = getelementptr inbounds i32* %b_load, i32 9
  store i32 3, i32* %b_ptr_index15
  %b_ptr_index16 = getelementptr inbounds i32* %b_load, i32 10
  store i32 2, i32* %b_ptr_index16
  %b_ptr_index17 = getelementptr inbounds i32* %b_load, i32 11
  store i32 1, i32* %b_ptr_index17
  %a = load i32** @a
  %b = load i32** @b
  %addMat = call i32 @addMat(i32* %a, i32* %b)
  %a18 = load i32** @a
  %printMatrix_result = call i32 @printMatrix(i32* %a18)
  %a19 = load i32** @a
  %b20 = load i32** @b
  %minusMat = call i32 @minusMat(i32* %a19, i32* %b20)
  %a21 = load i32** @a
  %printMatrix_result22 = call i32 @printMatrix(i32* %a21)
  ret i32 0
}

define i32 @printMatrix(i32* %mat) {
entry:
  %mat1 = alloca i32*
  store i32* %mat, i32** %mat1
  %mat_load = load i32** %mat1
  %mat_dim_2 = getelementptr inbounds i32* %mat_load, i32 2
  %mat_dim_2value_ = load i32* %mat_dim_2
  %mat_dim_1 = getelementptr inbounds i32* %mat_load, i32 1
  %mat_dim_1value_ = load i32* %mat_dim_1
  %tmp3_ = mul i32 1, %mat_dim_2value_
  %tmp_ = mul i32 %tmp3_, 0
```

```
  %tmp2_ = add i32 3, %tmp_
  %tmp3_2 = mul i32 %tmp3_, %mat_dim_1value_
  %mat_element = getelementptr inbounds i32* %mat_load, i32 %tmp2_
  %mat_element3 = load i32* %mat_element
  %printf = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([4
   ↪  x i8]* @fmt1, i32 0, i32 0), i32 %mat_element3)
  ret i32 0
}

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

declare i32 @copyMat(i32*, i32*)

declare i32 @minusMat(i32*, i32*)

declare i32 @addMat(i32*, i32*)

declare void @free(i8*)

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

7. Lessons Learnt

7.1 Abhishek

Designing and implementing a compiler is a challenging but fulfiling 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 langauge 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

```
(* MatCV scanner *)

{ open Parser }

rule token = parse
|  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment 0 lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LSQBRACKET }
| ']'      { RSQBRACKET }
| ':'      { COLON }
| ';'      { SEMI }
| ','      { COMMA }
| '.'      { DOT }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| "+."     { MATPLUS }
| "-."     { MATMINUS }
| "*."     { MATTIMES }
| "/."     { MATDIVIDE }
| '%'      { MOD }
| '^'      { EXP }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
```

```
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "row"     { ROW }
| "col"     { COL }
| "ele"     { ELE }
| "pixel"    { PIXEL }
| "var"     { VARKEYWORD }
| "const"     { CONSTANT }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "break"    { BREAK }
| "continue"    { CONTINUE }
| "exit"    { EXIT }
| "while"  { WHILE }
| "return" { RETURN }
| "function" { FUNCTION }
| "true"   { TRUE }
| "false"  { FALSE }
| ['0'-'9']+ as lexeme { LITERAL(int_of_string lexeme) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lexeme { ID(lexeme) }
| eof { EOF }
| _ as char { raise (Failure("Illegal Character: " ^ Char.escaped char)) }

(* For nested comments *)
and comment level = parse
| "/*" {comment (level + 1) lexbuf}
| "*/" {  if level = 0 then token lexbuf else comment (level - 1) lexbuf}
| _    { comment level lexbuf }
```

8.2 parser.mly

```
/* MatCV Parser */
%{ open Ast %}

/********* TODO ***********/
/**********MATRIX SPLICING? ******/
```

```
%token LPAREN  RPAREN  LBRACE RBRACE  LSQBRACKET  RSQBRACKET
%token COLON  SEMI  COMMA  DOT  PLUS  MINUS  TIMES  DIVIDE
%token MATPLUS  MATMINUS  MATTIMES  MATDIVIDE
%token MOD  EXP  ASSIGN  EQ  NEQ  LT  LEQ  GT  GEQ  AND  OR
%token NOT  ROW  COL  ELE PIXEL  VARKEYWORD  CONSTANT
%token IF  ELSE  FOR  BREAK  CONTINUE  EXIT  WHILE  RETURN
%token FUNCTION  TRUE  FALSE  EOF
%token <int> LITERAL
%token <string> ID

%left SEMI
%left COMMA
%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS MATPLUS MATMINUS
%left TIMES DIVIDE MOD MATTIMES MATDIVIDE
%right NOT NEG
%right EXP
%left DOT
%nonassoc UNBOUNDED



%start program
%type <Ast.program> program


%%

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

statements:
    /*nothing*/ { [], [] }
    |statements statement { ($2 :: fst $1), snd $1 }
```

```
    |statements functionDefinition { fst $1, ($2 :: snd $1) }

statementList:
/*nothing*/ { [] }
| statementList statement { $2 :: $1 }


functionDefinition:
FUNCTION ID LPAREN formalArguments RPAREN LBRACE statementList RBRACE {
    {fname = $2;
         formals = List.rev $4;
         body = List.rev $7}}

ifStatement:
IF LPAREN expr RPAREN statement %prec NOELSE { If($3, $5, Block([]))}
| IF LPAREN expr RPAREN statement ELSE statement { If($3, $5, $7) }

blockOfStatements:
LBRACE statementList RBRACE { Block(List.rev $2) }

forLoop:
FOR LPAREN optionalVarAssign SEMI optionalExpression SEMI
 ↪  optionalVarAssign RPAREN statement { For($3, $5, $7, $9) }


optionalVarAssign:
    /*nothing*/ { Nodecl }
| variableDeclaration {$1}

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


whileLoop:
WHILE LPAREN expr RPAREN statement { While($3, $5) }


statement:
    /*nothing*/
| blockOfStatements                          { $1 }
| expr SEMI                                    { Expr $1 }
| variableDeclaration SEMI          { VarDecl $1 }
| returnStatement  SEMI              {$1}
```

```
| ifStatement                               {$1}
| forLoop                                       {$1}
| whileLoop                                       {$1}
| rowLoop                                       {$1}
| eleLoop                                       {$1}
| pixelLoop                                       {$1}
| EXIT SEMI                             {Exit}
| BREAK SEMI                        {Break}
| CONTINUE SEMI                   {Continue}


eleLoop:
    ELE ID COLON ID statement { ForEachLoop ($2, $4, $5, Ele) }

rowLoop:
    ROW ID COLON ID statement { ForEachLoop ($2, $4, $5, Row) }

pixelLoop:
    PIXEL ID COLON ID statement { ForEachLoop ($2, $4, $5, Pixel) }


actualArguments:
/*nothing*/                    {[]}
| expr                { [$1] }
| actualArguments COMMA expr { $3 :: $1 }

formalArguments:
/*nothing*/ { [] }
| ID  { [$1] }
| formalArguments COMMA ID { $3 :: $1 }

returnStatement:
RETURN expr { Return($2) }
| RETURN { Return(Noexpr) }


variableDeclaration:
ID ASSIGN LBRACE matrixInitValues RBRACE { Matrix($1, List.rev (List.rev (fst $4) :: snd $4)) }
| ID ASSIGN dimensions { DimAssign($1, List.rev $3) }
| ID ASSIGN expr { ExprAssign($1,$3) }
| ID dimensions ASSIGN expr { MatElementAssign($1, List.rev $2, $4) }
```

```
dimensions:
    LSQBRACKET expr RSQBRACKET { [$2] }
| dimensions LSQBRACKET expr RSQBRACKET {$3 :: $1}


matrixInitValues:
| matrixInitValues COMMA expr {  $3 :: fst $1, snd $1 }
| matrixInitValues SEMI expr  {  [$3],   List.rev (fst $1) :: snd $1}
| expr { [$1], [] }

functionCall:
ID LPAREN actualArguments RPAREN {Call($1, List.rev $3)}

expr:
    LITERAL  { Literal($1) }
  | TRUE        { BoolLit(true) }
  | FALSE        { BoolLit(false) }
  | ID      { Id($1) }
  | ID MATPLUS ID     { MatPlus($1, $3) }
  | ID MATMINUS ID     { MatMinus($1, $3) }
  | LT ID COMMA expr GT %prec UNBOUNDED { UnboundedAccessRead($2, $4) }
  | LSQBRACKET LSQBRACKET ID COMMA expr COMMA expr RSQBRACKET
   ↪  RSQBRACKET %prec UNBOUNDED{ UnboundedAccessWrite($3, $5, $7) }
  | ID dimensions { MatAccess($1, List.rev $2) }
  | expr PLUS   expr { BinaryOp($1, Add,   $3) }
  | expr MINUS  expr { BinaryOp($1, Sub,   $3) }
  | expr TIMES  expr { BinaryOp($1, Mul,   $3) }
  | expr DIVIDE expr { BinaryOp($1, Div,   $3) }
  | expr MOD    expr { BinaryOp($1, Mod,   $3) }
  | expr EQ     expr { BinaryOp($1, Equal,   $3) }
  | expr NEQ    expr { BinaryOp($1, Neq,   $3) }
  | expr LT     expr { BinaryOp($1, Less,   $3) }
  | expr LEQ    expr { BinaryOp($1, Leq,   $3) }
  | expr GT     expr { BinaryOp($1, Greater,   $3) }
  | expr GEQ    expr { BinaryOp($1, Geq,   $3) }
  | expr AND    expr { BinaryOp($1, And,   $3) }
  | expr OR     expr { BinaryOp($1, Or,   $3) }
  | expr EXP    expr { BinaryOp($1, Exp,   $3) }
  | MINUS expr %prec NEG { Unop(Neg, $2) }
  | NOT expr  { Unop(Not, $2) }
  | functionCall {$1}
  | LPAREN expr RPAREN {$2}
```

8.3 ast.ml

```
(* MatCV AST *)

(* TODO *)

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

type uop = Neg | Not

type loopType = Row | Ele | Pixel

type expr =
    Literal of int
  | BoolLit of bool
  | Id of string
  | UnboundedAccessRead of string * expr
  | UnboundedAccessWrite of string * expr * expr
  | MatPlus of string * string
  | MatMinus of string * string
  | MatAccess of string * expr list
  | BinaryOp of expr * op * expr
  | Unop of uop * expr
  | Call of string * expr list
  | Noexpr


type varDecl =
    | Nodecl
    | Matrix of string * expr list list
    | ExprAssign of string * expr
    | DimAssign of string * expr list
    | MatElementAssign of string * expr list * expr


type statement =
    | Block of statement list
    | Expr of expr
    | VarDecl of varDecl
    | Return of expr
    | For of varDecl * expr * varDecl * statement
    | While of expr * statement
    | If of expr * statement * statement
    | Exit
```

```
    | Break
    | ForEachLoop of string * string * statement * loopType
    | Continue

type functionDefinition = {
    fname: string;
    formals: string list;
    body: statement list
}

type program = statement list * functionDefinition list

(* Supported Types *)
type builtInType =
    | Void (* For things that don't return anything *)
    | Int
    | Bool
    | Mat of int
    | Annotation of string
    | Func
    (* returnType * formalType list *)
    | FuncSignature of builtInType * builtInType list
    | Empty
    | Keyword

(* Annotated Expression*)
type aexpr =
    ALiteral of int * builtInType
  | ABoolLit of bool * builtInType
  | AId of string * builtInType
  | AUnboundedAccessRead of string * builtInType * aexpr * builtInType
  | AUnboundedAccessWrite of string * builtInType * aexpr * aexpr * builtInType
  | AMatPlus of string * builtInType * string * builtInType * builtInType
  | AMatMinus of string * builtInType * string * builtInType * builtInType
  | AMatAccess of string * builtInType * aexpr list * int * builtInType
  | ABinaryOp of aexpr * op * aexpr * builtInType
  | AUnop of uop * aexpr * builtInType
  | ACall of string * builtInType * aexpr list * builtInType
  | ANoexpr of builtInType


(* Annotated variable declarations *)
type avarDecl =
    | AMatrix of string * builtInType * aexpr list list * int * int * builtInType
```

```
    | AExprAssign of string * builtInType * aexpr * builtInType
    | ADimAssign of string * builtInType * aexpr list * int * builtInType
    | AMatElementAssign of string * builtInType * aexpr list * aexpr * int * builtInT
    | ANodecl of builtInType


(* Annotated statements *)
type astatement =
    | ABlock of astatement list * builtInType
    | AExpr of aexpr * builtInType
    | AVarDecl of avarDecl * builtInType
    | AReturn of builtInType * aexpr  * builtInType
    | AFor of avarDecl * aexpr * avarDecl * astatement * builtInType
    | AWhile of aexpr * astatement * builtInType
    | AIf of aexpr * astatement * astatement * builtInType
    | AExit of builtInType
    | ABreak of builtInType
    | AForEachLoop of string * builtInType * string * builtInType * astatement * loop
    | AContinue of builtInType

(* Annotated functions *)

type afunctionDefinition = {
    afname: string * builtInType;
    aformals: (string * builtInType) list;
    abody: astatement list;
    retType: builtInType
}


(* Pretty Printing function *)

let rec string_of_builtInType = function
    | Void -> "Void"
    | Int -> "Int"
    | Bool -> "Bool"
    | Mat(nDims) -> "Mat(" ^ string_of_int nDims ^ ")"
    | Annotation(m) -> m
    | Func -> "Func"
    | Empty -> "Empty"
    | Keyword -> "Keyword"
    | FuncSignature(returnType, formalTypeList) -> "FuncSignature: ReturnType: " ^ str
```

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

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

let string_of_boolLit = function
    true -> "true"
  | false -> "false"


let rec string_of_aexpr = function
  | ALiteral(l, t) ->  string_of_int l
  | ABoolLit(b, t) ->  string_of_boolLit b
  | AId(s, t) -> s
  | AUnboundedAccessRead(id, _, expr, _) -> "<" ^id ^ "," ^ string_of_aexpr expr ^ ">"
  | AUnboundedAccessWrite(id, _, expr1, expr2, _) -> "[[" ^ id ^ "," ^ string_of_aexpr
  | AMatPlus(id1,_,id2,_,_) -> id1 ^ " +. " ^ id2
  | AMatMinus(id1,_,id2,_,_) -> id1 ^ " -. " ^ id2
  | AMatAccess(id, t1, exprLst,i, t2) -> id ^  string_of_dim exprLst
  | ABinaryOp(e1, o, e2, t) ->
      string_of_aexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_aexpr e2
  | AUnop(o, e, t) ->  string_of_uop o ^ string_of_aexpr e
  | ACall(f, t1, el, t2) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_aexpr el) ^ ")"
  | ANoexpr(t) -> ""
```

```
  and string_of_dim = function
| [] -> ""
| a::lst -> "[" ^ string_of_aexpr a ^ "]" ^ string_of_dim lst

let rec string_of_row = function
  | [] -> ""
  | [a] -> string_of_aexpr a
  | a::lst-> string_of_aexpr a ^ "," ^ string_of_row lst


let rec string_of_mat = function
  | [] -> ""
  | [a] ->  string_of_row  a
  | a::lst-> string_of_row a  ^ ";" ^ string_of_mat lst


let rec string_of_avarDecl = function
    | AMatrix(matrixName, t1, mat, i1, i2, t2) -> "\n" ^ string_of_builtInType t1 ^ "
    | AExprAssign(id, t1,  expr, t2) ->  string_of_builtInType t1 ^ " " ^  id ^ " = "
    | ADimAssign(id, t1, expr, i, t2) ->    string_of_builtInType t1   ^ " " ^ id ^ "
    | AMatElementAssign(s, t1 , aexpr1, aexpr2, i, t2) -> string_of_builtInType t1 ^ "
    | ANodecl(t) -> ""

let string_of_loopType = function
| Row -> "row"
| Ele -> "ele"
| Pixel -> "pixel"


let rec string_of_aforDecl = function
| AExprAssign(id,t1, expr, t2) -> string_of_builtInType t1 ^ " " ^ id ^ " = " ^ string
| AMatrix(matrixName, t1, mat, i1, i2, t2) ->  matrixName ^ " = {" ^ string_of_mat mat
| ADimAssign(id, t1, expr,i , t2) -> string_of_builtInType t1   ^ " " ^  id ^ " = "
| AMatElementAssign(s, t1 , aexpr1, aexpr2, i, t2) -> string_of_builtInType t1 ^ " " ^
| ANodecl(t) -> ""




let rec string_of_astmt = function
    ABlock(stmts, t) ->
      "{\n" ^ String.concat "" (List.map string_of_astmt stmts)  ^ "}\n"
  | AExpr(expr, t) -> string_of_aexpr expr ^ ";\n";
  | AVarDecl(varDecl, t) ->   string_of_avarDecl varDecl ^ "\n";
```

```
  | AReturn(_, expr, t) -> "return " ^ string_of_aexpr expr ^ ";\n";
  | AIf(e, s1, ABlock([], Void), t) -> "if (" ^ string_of_aexpr e ^ ")\n" ^ string_of_
  | AIf(e, s1, s2, t) ->  "if (" ^ string_of_aexpr e ^ ")\n" ^
      string_of_astmt s1 ^ "else\n" ^ string_of_astmt s2
  | AFor(v1, e2, v2, s, t) ->
     "for (" ^ string_of_aforDecl v1  ^ ";" ^ string_of_aexpr e2 ^ ";" ^
      string_of_aforDecl v2  ^ ") " ^ string_of_astmt s
  | AWhile(e, s, t) ->  "while (" ^ string_of_aexpr e ^ ") " ^ string_of_astmt s
  | AForEachLoop(str1,t1,str2,t2,s1,loopT,t3) ->  "for " ^ string_of_loopType loopT ^
  | AExit(t) -> "exit; \n"
  | ABreak(t) -> "break; \n"
  | AContinue(t) -> "continue; \n"




let rec  string_of_aformals = function
  | [] -> ""
  | [(s,t)] ->  string_of_builtInType t ^ " " ^ s
  | (s,t)::lst-> string_of_builtInType t ^ " " ^ s ^ "," ^ string_of_aformals lst



let string_of_afname = function

  | (s,t) -> string_of_builtInType t^ "_" ^ s


let string_of_afdecl fdecl =
  string_of_builtInType fdecl.retType ^ " " ^ string_of_afname fdecl.afname ^ "("  ^ (
  ")\n{\n" ^
  String.concat "" (List.map string_of_astmt fdecl.abody) ^
  "}\n"


let string_of_program (stmnt, funcs) =
  String.concat "" (List.map string_of_astmt stmnt) ^ "\n" ^
  String.concat "\n" (List.map string_of_afdecl funcs)
```

8.4 semant.ml

```
(* MatCV Semantic Checker *)

open Ast
```

```ocaml

(*module ReservedWords = Set.Make(String)*)

let keywords = ["row"; "col"; "ele"; "pixel"; "var"; "const"; "if"; "else"; "for"; "break"; "continue"; "exit"; "while"; "return"; "function"; "true"; "false"]

let builtInFunctions = ["print"; "main"]

let code = ref ([])

let getListSize lst = List.fold_left (fun acc _ -> acc + 1) 0 lst

let generateTypeForAnnotation() =
let rec genHelp = function
| 'z'::tail -> 'a' :: List.rev (genHelp (List.rev tail))
| c::tail -> (Char.chr ((Char.code c) + 1)) :: tail
| [] -> ['a']
in
let updatedCode = genHelp !code
in code := updatedCode;
Annotation(String.concat "" (List.map Char.escaped updatedCode));;


let typeOfAexpr = function
| ALiteral(_, t) -> t
| ABoolLit(_, t) -> t
| AId(_, t) -> t
| AMatPlus (_, _, _, _, t)|AMatMinus (_, _, _, _, t) -> t
| AUnboundedAccessRead( _, _, _, t) -> t
| AUnboundedAccessWrite(_, _, _, _, t) -> t
| AMatAccess(_, _, _, _, t) -> t
| ABinaryOp(_, _, _, t) -> t
| AUnop(_, _, t) -> t
| ACall(_, _, _, t) -> t
| ANoexpr(t) -> t


let typeOfAvarDecl = function
| ANodecl(t) -> t
| AMatrix(_, _, _, _, _, t) -> t
| AExprAssign(_, _, _, t) -> t
| ADimAssign(_, _, _, _, t) -> t
| AMatElementAssign(_, _, _, _, _, t) -> t
```

```
let typeOfStatement = function
| ABlock(_, t) -> t
| AExpr(_, t) -> t
| AVarDecl(_, t) -> t
| AReturn(_, _, t) -> t
| AFor(_, _, _, _, t) -> t
| AWhile(_, _, t) -> t
| AIf(_, _, _, t) -> t
| AExit(t) -> t
| ABreak(t) -> t
| AForEachLoop(_, _, _, _, _, _, t) -> t
| AContinue(t) -> t

(* Old Code *)
let getVariableDeclFromStatement statements =
    let rec helper acc = function
    | [] -> acc
    | VarDecl(s) :: t -> helper (VarDecl(s)::acc) t
    | _ :: t -> helper acc t
    in helper [] statements


(* Handle errors *)
let printError message =
    print_string("\nError: " ^ message); exit 1

let printWarning message =
    print_string ("\nWarning: " ^ message)

let printReservedError name =  printError ("Name: " ^ name ^ " is reserved.")

 let printDuplicateFunctionError typ m =
     match typ with
             | Keyword -> printReservedError m
             | _ -> printError ("Multiple definitions of function: " ^ m)

let printUndefinedVariableError m = printError ("Undefined variable: " ^ m)
```

```
let printInvalidDimensionsError m = printError ("Invalid dimensions were specified for Matrix: " ^ m)

let printTypeMismatchError id t1 t2 = printError ("Type Mismatch: Cannot assign type " ^ (string_of_builtInType t2) ^ " to " ^ id ^ ". Previously had type: " ^ (string_of_builtInType t1) )


let rec annotateExpression globalSymbolTable localSymbolTable = function
  | Literal(l) -> ALiteral(l, Int)
  | BoolLit(b) -> ABoolLit(b, Bool)
  | Id(id) -> let idType = if Hashtbl.mem localSymbolTable id
                           then Hashtbl.find localSymbolTable id
                           else if Hashtbl.mem globalSymbolTable id
                           then Hashtbl.find globalSymbolTable id
                           else let _ = printUndefinedVariableError id in Void in
                           AId(id, idType)
  | MatPlus(id1, id2) ->
                           let idType1 = if Hashtbl.mem localSymbolTable id1
                           then Hashtbl.find localSymbolTable id1
                           else if Hashtbl.mem globalSymbolTable id1
                           then Hashtbl.find globalSymbolTable id1
                           else let _ = printUndefinedVariableError id1 in Void in

                           let idType2 = if Hashtbl.mem localSymbolTable id2
                           then Hashtbl.find localSymbolTable id2
                           else if Hashtbl.mem globalSymbolTable id2
                           then Hashtbl.find globalSymbolTable id2
                           else let _ = printUndefinedVariableError id2 in Void in
                           AMatPlus(id1, idType1, id2, idType2, generateTypeForAnnotation())

  | MatMinus(id1, id2) ->
                           let idType1 = if Hashtbl.mem localSymbolTable id1
                           then Hashtbl.find localSymbolTable id1
                           else if Hashtbl.mem globalSymbolTable id1
                           then Hashtbl.find globalSymbolTable id1
```

```
                        else let _ = printUndefinedVariableError id1 in Void in

                        let idType2 = if Hashtbl.mem localSymbolTable id2
                        then Hashtbl.find localSymbolTable id2
                        else if Hashtbl.mem globalSymbolTable id2
                        then Hashtbl.find globalSymbolTable id2
                        else let _ = printUndefinedVariableError id2 in Void in
                        AMatMinus(id1, idType1, id2, idType2, generateTypeForAnnotation())

    | UnboundedAccessRead(id, expr) -> let idType = if Hashtbl.mem localSymbolTable id
                        then Hashtbl.find localSymbolTable id
                        else if Hashtbl.mem globalSymbolTable id
                        then Hashtbl.find globalSymbolTable id
                        else let _ = printUndefinedVariableError id in Void in
                        let aexpr = annotateExpression globalSymbolTable localSymbolTable expr in
                        AUnboundedAccessRead(id, idType, aexpr, generateTypeForAnnotation())

    | UnboundedAccessWrite(id, expr1, expr2) -> let idType = if Hashtbl.mem localSymbolTable id
                        then Hashtbl.find localSymbolTable id
                        else if Hashtbl.mem globalSymbolTable id
                        then Hashtbl.find globalSymbolTable id
                        else let _ = printUndefinedVariableError id in Void in
                        let aexpr1 = annotateExpression globalSymbolTable localSymbolTable expr1 in
                        let aexpr2 = annotateExpression globalSymbolTable localSymbolTable expr2 in
                        AUnboundedAccessWrite(id, idType, aexpr1, aexpr2, generateTypeForAnnotation())

    | MatAccess(id, exprList) -> let idType =
                            if Hashtbl.mem localSymbolTable id then
```

```
                            Hashtbl.find localSymbolTable id
                            else if Hashtbl.mem globalSymbolTable id then
                            Hashtbl.find globalSymbolTable id
                            else let _ = printUndefinedVariableError id in Void in
                            let aExprList = List.map (fun expr -> annotateExpression globalSymbolTable localSymbolTable expr) exprList in
                            let nDimensions = getListSize exprList in
                            AMatAccess(id, idType, aExprList, nDimensions, generateTypeForAnnotation())

    | BinaryOp(expr1, op, expr2) -> let aexpr1 = annotateExpression globalSymbolTable localSymbolTable expr1 in
                                    let aexpr2 = annotateExpression globalSymbolTable localSymbolTable expr2 in
                                    ABinaryOp(aexpr1, op, aexpr2, generateTypeForAnnotation())

    | Unop(uop, expr) -> let aexpr = annotateExpression globalSymbolTable localSymbolTable expr in
                                    AUnop(uop, aexpr, generateTypeForAnnotation())

    | Call(id, exprList) -> let idType =
                        if Hashtbl.mem localSymbolTable id then
                        Hashtbl.find localSymbolTable id
                        else if Hashtbl.mem globalSymbolTable id then
                        Hashtbl.find globalSymbolTable id
                        else let _ = printUndefinedVariableError id in Void in
                        let aExprList = List.map (fun expr -> annotateExpression globalSymbolTable localSymbolTable expr) exprList in
                        ACall(id, idType, aExprList, generateTypeForAnnotation())

    | Noexpr -> ANoexpr(Void)
```

```
let rec annotateVarDecl globalSymbolTable localSymbolTable = function
      | Nodecl -> ANodecl(Void)
      | Matrix(id, exprListList) -> let idType =
                                if Hashtbl.mem localSymbolTable id then
                                Hashtbl.find localSymbolTable id
                                else if Hashtbl.mem globalSymbolTable id then
                                Hashtbl.find globalSymbolTable id
                                else let idt = generateTypeForAnnotation() in
                                (* Add a generated type to the local symbol table *)
                                let _ = Hashtbl.add localSymbolTable id idt
                                in idt
                                in
                                let _ = if idType = Keyword then printReservedError id
                                in
                                let nRows = getListSize exprListList
                                in
                                let nCols = if nRows <> 0 then
                                    getListSize (List.hd exprListList)
                                    else 0
                                in
                                (* Check whether all rows have equal number of elements *)
                                let _ = List.iter (fun exprList -> if (getListSize exprList) <> nCols then
                                    printInvalidDimensionsError id) exprListList
                                in
                                (* Annotate each element *)
                                let aExprListList = List.map
                                (fun exprList -> List.map (fun expr ->
                                    annotateExpression globalSymbolTable localSymbolTable expr)
```

```
                                                        exprList) exprListList in
                          (* Store the row and column count with this matrix *)
                          (* Will help in code generation *)
                          AMatrix(id, idType, aExprListList, nRows, nCols, Void)

    | ExprAssign(id, expr) -> let idType =
                          if Hashtbl.mem localSymbolTable id then
                          Hashtbl.find localSymbolTable id
                          else if Hashtbl.mem globalSymbolTable id then
                          Hashtbl.find globalSymbolTable id
                          else let idt = generateTypeForAnnotation() in
                          (* Add a generated type to the local symbol table *)
                          let _ = Hashtbl.add localSymbolTable id idt
                          in idt
                          in
                          let _ = if idType = Keyword then printReservedError id
                          in
                          let aExpr = annotateExpression globalSymbolTable localSymbolTable expr
                          in
                          AExprAssign(id, idType, aExpr, Void)

    | DimAssign(id, exprList) -> let idType =
                          if Hashtbl.mem localSymbolTable id then
                          Hashtbl.find localSymbolTable id
                          else if Hashtbl.mem globalSymbolTable id then
                          Hashtbl.find globalSymbolTable id
                          else let idt = generateTypeForAnnotation() in
                          (* Add a generated type to the local symbol table *)
                          let _ = Hashtbl.add localSymbolTable id idt
```

```
                        in idt
                        in
                        let _ = if idType = Keyword then printReservedError id
                        in
                        let aExprList = List.map (fun expr ->
                                                annotateExpression globalSymbolTable localSymbolTable expr)
                                                exprList
                        in
                        let nDimensions = getListSize exprList in
                        ADimAssign(id, idType, aExprList, nDimensions, Void)

  | MatElementAssign(id, exprList, expr) -> let idType =
                        if Hashtbl.mem localSymbolTable id then
                        Hashtbl.find localSymbolTable id
                        else if Hashtbl.mem globalSymbolTable id then
                        Hashtbl.find globalSymbolTable id
                        else
                        let _= printUndefinedVariableError id in
                        Empty
                        in
                        let _ = if idType = Keyword then printReservedError id
                        in
                        let aExprList = List.map (fun expr ->
                                                annotateExpression globalSymbolTable localSymbolTable expr)
                                                exprList
                        in
                        let aExpr = annotateExpression globalSymbolTable localSymbolTable expr
                        in
```

```
                                let nDimensions = getListSize exprList in
                                AMatElementAssign(id, idType, aExprList, aExpr, nDimensions, Void)


let mergeSymbolTables globalSymbolTable localSymbolTable = let mergedSymbolTable = Hashtbl.create 100 in
          let _ = Hashtbl.iter (fun key value -> Hashtbl.add mergedSymbolTable key value) localSymbolTable
          in
          let _ = Hashtbl.iter (fun key value -> if not (Hashtbl.mem mergedSymbolTable key) then Hashtbl.add mergedSymbolTable key value) globalSymbolTable
          in mergedSymbolTable




let rec annotateStatement globalSymbolTable localSymbolTable ?isControlFlowAllowed:(isCFA = false) inFunction = function
    | Block (statementList) -> let newGlobalSymbolTable = mergeSymbolTables globalSymbolTable localSymbolTable in
                               let newLocalSymbolTable = Hashtbl.create 100 in
                               let aStatementList = List.map (fun statement -> annotateStatement newGlobalSymbolTable newLocalSymbolTable inFunction statement ~isControlFlowAllowed:isCFA) statementList
                               in
                               ABlock (aStatementList, Void)
    | Expr (expr) -> let aExpr = annotateExpression globalSymbolTable localSymbolTable expr in
                     AExpr(aExpr, Void)
    | VarDecl (varDecl) -> let aVarDecl = annotateVarDecl globalSymbolTable localSymbolTable varDecl in
                           AVarDecl(aVarDecl, Void)
    | Return (expr) -> let _ = if inFunction = "main" then printError "Cannot use return outside functions." in let aExpr = annotateExpression globalSymbolTable localSymbolTable expr in
```

```
    let funcType = Hashtbl.find globalSymbolTable inFunction
    in (match funcType with
    | FuncSignature(returnTypeSig, formalTypeList) -> AReturn (returnTypeSig, aExpr, Void)
    | _ -> let _ = printError "Invalid use of return statement." in AReturn (Void, aExpr, Void)
    )

    | For (varDecl1, expr, varDecl2, statement) -> let aVarDecl1 = annotateVarDecl globalSymbolTable localSymbolTable varDecl1 in
                                                   let aVarDecl2 = annotateVarDecl globalSymbolTable localSymbolTable varDecl2 in
                                                   let aExpr = annotateExpression globalSymbolTable localSymbolTable expr in
                                                   let aStatement = annotateStatement globalSymbolTable localSymbolTable ~isControlFlowAllowed:true inFunction statement  in
                                                   AFor (aVarDecl1, aExpr, aVarDecl2, aStatement, Void)
    | While (expr, statement) -> let aExpr = annotateExpression globalSymbolTable localSymbolTable expr in
                                 let aStatement = annotateStatement globalSymbolTable localSymbolTable ~isControlFlowAllowed:true inFunction statement in
                                 AWhile (aExpr, aStatement, Void)
    | If (expr, statement1, statement2) ->
            let newGlobalSymbolTable = mergeSymbolTables globalSymbolTable localSymbolTable in
            let newLocalSymbolTable = Hashtbl.create 100 in
```

```
            let aExpr = annotateExpression newGlobalSymbolTable newLocalSymbolTable expr in
                                              let aStatement1 = annotateStatement newGlobalSymbolTable newLocalSymbolTable ~isControlFlowAllowed:isCFA inFunction statement1 in
                                              let aStatement2 = annotateStatement newGlobalSymbolTable newLocalSymbolTable ~isControlFlowAllowed:isCFA inFunction statement2 in
                                              AIf(aExpr, aStatement1, aStatement2, Void)
    | Exit -> AExit(Void)
    | Break -> let _ = if not isCFA then printError "Invalid use of break." in ABreak(Void)
    | ForEachLoop (id, objName, statement, loopType) ->
            let newGlobalSymbolTable = mergeSymbolTables globalSymbolTable localSymbolTable in
            let newLocalSymbolTable = Hashtbl.create 100 in
            let idType =
                        if Hashtbl.mem newLocalSymbolTable id then
                        Hashtbl.find newLocalSymbolTable id
                        else if Hashtbl.mem newGlobalSymbolTable id then
                        Hashtbl.find newGlobalSymbolTable id
                        else let idt = generateTypeForAnnotation() in
                        (* Add a generated type to the local symbol table *)
                        let _ = Hashtbl.add newLocalSymbolTable id idt
                        in idt
                        in
                        let objType =
                        if Hashtbl.mem newLocalSymbolTable objName then
                        Hashtbl.find newLocalSymbolTable objName
                        else if Hashtbl.mem newGlobalSymbolTable objName then
                        Hashtbl.find newGlobalSymbolTable objName
                        else
                        let _= printUndefinedVariableError objName in
                        Empty
```

```
                    in
                    let aStatement = annotateStatement newGlobalSymbolTable newLocalSymbolTable ~isControlFlowAllowed:true inFunction statement
                    in
                    AForEachLoop (id, idType, objName, objType, aStatement, loopType, Void)
    | Continue -> let _ = if not isCFA then printError "Invalid use of continue." in AContinue(Void)


let rec collectExpr = function
  |  ALiteral(_) | ABoolLit(_) | AId(_) | ANoexpr(_) -> []
  (* If someone accesses a variable like matrix, it means that id's
   * type should be Mat and each expression should evaluate to Int and
   * this expression returns an Int *)
  | AUnboundedAccessRead(id, idType, aexpr, exprType) ->  let constraints = [(exprType, Int)]
  in let exprConstr = collectExpr aexpr in [(typeOfAexpr aexpr, Int)] @ constraints @ exprConstr

  | AUnboundedAccessWrite(id, idType, aexpr1, aexpr2, exprType) -> let constraints = [(exprType, Int)]
  in let exprConstr1 = collectExpr aexpr1
  in let exprConstr2 = collectExpr aexpr2 in [(typeOfAexpr aexpr1, Int);(typeOfAexpr aexpr2, Int)] @ constraints @ exprConstr1 @ exprConstr2

  |AMatPlus(id1, idType1, id2, idType2, exprType) -> [(exprType, idType1); (idType1, idType2)]

  |AMatMinus(id1, idType1, id2, idType2, exprType) -> [(exprType, idType1); (idType1, idType2)]

  | AMatAccess(id, idType, aExprList, nDim, exprType) ->
          let constraints = [(idType, Mat(nDim)); (exprType, Int)] (* Not supporting a matrix of functions for now *)
          in let exprConstraints = List.fold_left (fun constraintAcc expr -> let exprConstr = collectExpr expr in (typeOfAexpr expr, Int) :: exprConstr @ constraintAcc) [] aExprList
          in constraints @ exprConstraints
  (* Now in case of binary operators, the result can be bool if the
   * operators are comparison operators etc. *)
```

```
  | ABinaryOp(aexpr1, op, aexpr2, exprType) ->
        let t1 = typeOfAexpr aexpr1 in let t2 = typeOfAexpr aexpr2
        in
        let constraints = match op with
        | Equal | Neq | Less | Leq | Greater | Geq | And | Or ->
                [(t1, t2); (exprType, Bool)]
        | Add | Sub | Mul | Div | Exp | Mod -> [(t1, Int); (t2, Int); (exprType, Int)]
        in
        constraints @ (collectExpr aexpr1) @ (collectExpr aexpr2)

  | AUnop(uop, aexpr, exprType) ->
        let t = typeOfAexpr aexpr in
        let constraints = match uop with
        | Neg -> [(t, Int);(exprType, Int)]
        | Not -> [(exprType, Bool)]
        in
        constraints @ (collectExpr aexpr)

        (* TODO: Add more constraints using function definition: *)
  | ACall(id, idType, aExprList, exprType) ->
        match idType with
        | FuncSignature(returnTypeSig, formalTypeList) ->
        let exprConstraints = List.fold_left (fun constraintAcc expr -> let exprConstr = collectExpr expr in exprConstr @ constraintAcc) [] aExprList
        in let size1 = getListSize formalTypeList in let size2 = getListSize aExprList in
        let _ = if size1 != size2 then printError ("Function: " ^ id ^ " called with: " ^ string_of_int size2 ^ " arguments. While the function expects: " ^ string_of_int size1 ^ " arguments.")
        in let formalConstraints = List.map2 (fun typ1 aExpr -> (typ1, typeOfAexpr(aExpr))) formalTypeList aExprList
        in let retConstraint = [(returnTypeSig, exprType)]
        in retConstraint @ formalConstraints @ exprConstraints
        | Keyword when id = "print" -> []
        | _ -> let _ = printError "Invalid use of function call." in []



let rec collectVarDecl = function
    | ANodecl(_) -> []
```

```
    | AMatrix(id, idType, aExprListList, nRows, nCols, _) ->
            let constraints = [(idType, Mat(2))]
            in
            let aExprListListConstraints = List.fold_left (fun
             ↪  constraintList aExprList ->
                List.fold_left (fun constrList aexpr -> (typeOfAexpr
                 ↪  aexpr, Int) ::(collectExpr aexpr) @ constrList)
                 ↪  constraintList aExprList
            ) [] aExprListList
            in constraints @ aExprListListConstraints

    | AExprAssign(id, idType, aExpr, _) ->
            let constraints = [(idType, typeOfAexpr aExpr)]
            in
            (collectExpr aExpr) @ constraints

    | ADimAssign(id, idType, aExprList, nDimensions,  _) ->
            let constraints = [(idType, Mat(nDimensions))]
            in
            let aExprListConstraints = List.fold_left (fun constrList
             ↪  expr -> (typeOfAexpr expr, Int) ::(collectExpr expr) @
             ↪  constrList) [] aExprList
            in constraints @ aExprListConstraints
    | AMatElementAssign(id, idType, aExprList, aExpr, nDimensions, _)
     ↪  ->
            let constraints = [(idType, Mat(nDimensions)); (typeOfAexpr
             ↪  aExpr, Int)]
            in
            let aExprListConstraints = List.fold_left (fun constrList
             ↪  expr -> (typeOfAexpr expr, Int) ::(collectExpr expr) @
             ↪  constrList) [] aExprList
            in constraints @ aExprListConstraints @ (collectExpr aExpr)



            (* All statements have type Void *)
let rec collectStatement = function
    | AContinue(_) | ABreak(_) | AExit(_) -> []

    | ABlock (aStatementList, _) -> List.fold_left (fun constraintAcc
     ↪  astatement -> (collectStatement astatement) @ constraintAcc) []
     ↪  aStatementList

```

```
    | AExpr(aExpr, _) -> collectExpr aExpr

    | AVarDecl(aVarDecl, _) -> collectVarDecl aVarDecl

    (* TODO: Relate return type to the annotated function *)
    | AReturn (retType, aExpr, _) -> [(retType, typeOfAexpr(aExpr))] @ (collectExpr aExpr)

    | AFor (aVarDecl1, aExpr, aVarDecl2, aStatement, _) -> let constLst1 = collectVarDecl aVarDecl1
                                                            in
                                                            let constLst2 = collectVarDecl aVarDecl2
                                                            in
                                                            let constLst3 = collectExpr aExpr
                                                            in
                                                            let constLst4 = collectStatement aStatement
                                                            in
                                                            (typeOfAexpr aExpr, Bool) :: constLst1 @ constLst2 @ constLst3 @ constLst4

    | AIf(aExpr, aStatement1, aStatement2, _) -> let constraints = (typeOfAexpr aExpr, Bool) :: (collectExpr aExpr)
                                                 in
```

```
                                        let constLst1 = collectStatement aStatement1
                                        in
                                        let constLst2 = collectStatement aStatement2
                                        in
                                        constraints @ constLst1 @ constLst2

    | AWhile (aExpr, aStatement, _) -> let constraints = (typeOfAexpr aExpr, Bool) :: (collectExpr aExpr)
                                  in
                                  let constLst = collectStatement aStatement
                                  in constraints @ constLst

    | AForEachLoop (id, idType, objName, objType, aStatement, loopType, _) -> let constraints = match loopType with
                                   | Row -> [(idType, Mat(2)); (objType, Mat(3))]
                                   | Ele -> [(idType, Int); (objType, Mat(3))]
                                   | Pixel -> [(idType, Mat(1)); (objType, Mat(3))]
                                   in
                                   constraints @ (collectStatement aStatement)


let rec substitute t1 t2 t =
    match t with
    | Void | Int | Bool | Func -> t
    | Mat(nDim) -> Mat(nDim)
    | Annotation(s) ->  if t1 = t then t2 else t
    | FuncSignature(_) -> Func
    | Keyword -> Keyword
    | _ -> printError "Unknown type error."

let apply substitutionList typ =
        List.fold_right (fun (t1, t2) t -> substitute t1 t2 t) substitutionList typ
```

```
let rec unifyOne s t =
  if s = t then (*let _ = print_string ("Unify one s = t:" ^ string_of_builtInType s ^ "," ^ string_of_builtInType t ^ "\n") in*) []
  else
      match (s, t) with
      | Annotation(x), Annotation(y)  -> [Annotation(x), Annotation(y)]
      | Annotation(x), y | y, Annotation(x) -> [(Annotation(x), y)]
      | x , y -> let _ = printError ("Mismatched types:" ^ string_of_builtInType x ^ "," ^ string_of_builtInType y ^ "\n") in []
and unify = function
  | [] -> []
  | (x, y) :: t ->
      let t2 = unify t in
      let t1 = unifyOne (apply t2 x) (apply t2 y) in
      t1 @ t2

let collectStatementList astatements = let constraints = List.fold_left (fun constraintList astatement -> (constraintList @ (collectStatement astatement))) [] astatements in constraints


let annotateFunctionHelper globalSymbolTable localSymbolTable func =
    let funcType = Hashtbl.find globalSymbolTable func.fname
    in match funcType with
    | FuncSignature(returnTypeSig, formalTypeList) ->
            let aFormals = List.map2 (fun id typ -> let _ =  (if Hashtbl.mem localSymbolTable id then printError ("Two or more formals have same name: " ^ id ^ " in function: " ^ func.fname)) in let _ = Hashtbl.add localSymbolTable id typ in (id,typ)) func.formals formalTypeList
            in
    {
        afname = (func.fname, funcType);
        aformals = aFormals;
        abody = List.map (fun statement -> annotateStatement globalSymbolTable localSymbolTable func.fname statement) func.body;
        retType = returnTypeSig;
    }
```

```
    | _ -> let _ = printError "Incorrect use of function: " ^ func.fname in
    (* Record shown below is useless. It is here to allow printError to work *)
    {
        afname = (func.fname, Void);
        aformals = List.map (fun id -> let _ = Hashtbl.add localSymbolTable id Void in (id,Void)) func.formals;
        abody = List.map (fun statement -> annotateStatement globalSymbolTable localSymbolTable func.fname statement) func.body;
        retType = Void;
    }



let annotateFunction globalSymbolTable func =
    let localSymbolTable = Hashtbl.create 100
    in
    annotateFunctionHelper globalSymbolTable localSymbolTable func



let collectFunction func = collectStatementList func.abody
let collectFunctionList functions = let constraints = List.fold_left (fun constraintList func -> (constraintList @ (collectFunction func))) [] functions in constraints


let rec applyExpression unifiedConstraints = function
  | ALiteral(_) as x -> x | ABoolLit(_) as x -> x | ANoexpr(_) as x -> x
  | AId(id, idType) -> AId(id, (apply unifiedConstraints idType))
  | AUnboundedAccessRead(id, idType, aexpr, exprType) -> AUnboundedAccessRead(id, (apply unifiedConstraints idType), (applyExpression unifiedConstraints aexpr), (apply unifiedConstraints exprType))
  | AUnboundedAccessWrite(id, idType, aexpr1, aexpr2, exprType) -> AUnboundedAccessWrite(id, (apply unifiedConstraints idType), (applyExpression unifiedConstraints aexpr1),(applyExpression unifiedConstraints aexpr2), (apply unifiedConstraints exprType))

```

```
  |AMatPlus(id1, idType1, id2, idType2, exprType) -> AMatPlus(id1, (apply unifiedConstraints idType1), id2, (apply unifiedConstraints idType2), (apply unifiedConstraints exprType))

  |AMatMinus(id1, idType1, id2, idType2, exprType) -> AMatMinus(id1, (apply unifiedConstraints idType1), id2, (apply unifiedConstraints idType2), (apply unifiedConstraints exprType))

  | AMatAccess(id, idType, aExprList, nDim, exprType) ->
        let resolvedExprList = List.map (fun aexpr -> applyExpression unifiedConstraints aexpr) aExprList
        in
        AMatAccess(id, (apply unifiedConstraints idType), resolvedExprList, nDim, (apply unifiedConstraints exprType))
  | ABinaryOp(aexpr1, op, aexpr2, exprType) ->
        ABinaryOp((applyExpression unifiedConstraints aexpr1), op, (applyExpression unifiedConstraints aexpr2), (apply unifiedConstraints exprType))

  | AUnop(uop, aexpr, exprType) -> AUnop(uop, applyExpression unifiedConstraints aexpr, apply unifiedConstraints exprType)
  | ACall(id, idType, aExprList, exprType) ->
        let resolvedExprList = List.map (fun aexpr -> applyExpression unifiedConstraints aexpr) aExprList
        in
        ACall(id, (apply unifiedConstraints idType), resolvedExprList, (apply unifiedConstraints exprType))



let rec applyVarDecl unifiedConstraints = function
    | ANodecl(_) as x -> x
    | AMatrix(id, idType, aExprListList, nRows, nCols, varDeclType) ->
        let resolveExpr = List.map (fun aexpr -> applyExpression unifiedConstraints aexpr)
        in
        let resolvedExprListList = List.map (fun aexprList -> resolveExpr aexprList) aExprListList
        in
        AMatrix(id, (apply unifiedConstraints idType), resolvedExprListList, nRows, nCols, (apply unifiedConstraints varDeclType))
    | AExprAssign(id, idType, aExpr, varDeclType) ->
```

```
        AExprAssign(id, (apply unifiedConstraints idType), (applyExpression unifiedConstraints aExpr), (apply unifiedConstraints varDeclType))
    | ADimAssign(id, idType, aExprList, nDimensions, varDeclType) ->
        let resolvedExprList = List.map (fun aexpr -> applyExpression unifiedConstraints aexpr) aExprList
        in
        ADimAssign(id, (apply unifiedConstraints idType), resolvedExprList, nDimensions, (apply unifiedConstraints varDeclType))
    | AMatElementAssign(id, idType, aExprList, aExpr, nDimensions, varDeclType) ->
        let resolvedExprList = List.map (fun aexpr -> applyExpression unifiedConstraints aexpr) aExprList
        in
        AMatElementAssign(id, (apply unifiedConstraints idType), resolvedExprList, (applyExpression unifiedConstraints aExpr), nDimensions, (apply unifiedConstraints varDeclType))


let rec applyStatement unifiedConstraints = function
    | AContinue(_) as x -> x | ABreak(_) as x -> x | AExit(_) as x -> x

    | ABlock (aStatementList, statementType) ->
        let resolvedStatementList = List.map (fun astatement -> applyStatement unifiedConstraints astatement) aStatementList
        in
        ABlock(resolvedStatementList, (apply unifiedConstraints statementType))

    | AExpr(aExpr, statementType) ->
        AExpr((applyExpression unifiedConstraints aExpr), (apply unifiedConstraints statementType))

    | AVarDecl(aVarDecl, statementType) ->
        AVarDecl((applyVarDecl unifiedConstraints aVarDecl), (apply unifiedConstraints statementType))

    | AReturn(returnType, aExpr, statementType) ->
        AReturn((apply unifiedConstraints returnType), (applyExpression unifiedConstraints aExpr), (apply unifiedConstraints statementType))
```

```
    | AFor(aVarDecl1, aExpr, aVarDecl2, aStatement, statementType) ->
         AFor((applyVarDecl unifiedConstraints aVarDecl1), (applyExpression unifiedConstraints aExpr), (applyVarDecl unifiedConstraints aVarDecl2), (applyStatement unifiedConstraints aStatement), (apply unifiedConstraints statementType))


    | AIf(aExpr, aStatement1, aStatement2, statementType) ->
         AIf((applyExpression unifiedConstraints aExpr), (applyStatement unifiedConstraints aStatement1), (applyStatement unifiedConstraints aStatement2), (apply unifiedConstraints statementType))

    | AWhile(aExpr, aStatement, statementType) ->
         AWhile((applyExpression unifiedConstraints aExpr), (applyStatement unifiedConstraints aStatement), (apply unifiedConstraints statementType))

    | AForEachLoop(id, idType, objName, objType, aStatement, loopType, statementType) ->
         AForEachLoop(id, (apply unifiedConstraints idType), objName, (apply unifiedConstraints objType), (applyStatement unifiedConstraints aStatement), loopType, (apply unifiedConstraints statementType))


let rec applyStatementList unifiedConstraints aStatementList = List.map (fun astatement -> applyStatement unifiedConstraints astatement) aStatementList



let applyFunction unifiedConstraints func =
    let (fname, _) = func.afname in
    {
        afname = (fname, Func);
        aformals = List.map (fun (id, typ) -> (id, (apply unifiedConstraints typ))) func.aformals;
        abody = applyStatementList unifiedConstraints func.abody;
        retType = apply unifiedConstraints func.retType(*let rType = (apply unifiedConstraints func.retType) in match rType with
```

```
                    | Annotation(_) -> Void
                    | x -> x*)
                    ;
    }

let applyFunctions unifiedConstraints functions =
    List.map (fun func -> applyFunction unifiedConstraints func)
     ↪  functions


    (* Check program semantics *)
let check_semantics (gstatements, functions) =
    let globalSymbolTable = Hashtbl.create 100 in
    let _ =
          List.iter (fun ele -> Hashtbl.add globalSymbolTable ele
           ↪  Keyword) (keywords @ builtInFunctions)
    in
    (* Check for duplicate functions *)
    let _ = List.iter (fun ele ->
            if Hashtbl.mem globalSymbolTable ele.fname then
            printDuplicateFunctionError (Hashtbl.find globalSymbolTable
             ↪  ele.fname) ele.fname
            else let formalTypes = List.map (fun _ ->
             ↪  generateTypeForAnnotation()) ele.formals
            in
            Hashtbl.add globalSymbolTable ele.fname
             ↪  (FuncSignature(generateTypeForAnnotation(),
             ↪  formalTypes))) functions
    in
    let localSymbolTable = Hashtbl.create 100 in
    let agstatements = List.map (fun statement -> annotateStatement
     ↪  globalSymbolTable localSymbolTable "main" statement)
     ↪  gstatements
    (* Overwrite globalSymbolTable with localSymbolTable *)
    in let globalSymbolTable = mergeSymbolTables globalSymbolTable
     ↪  localSymbolTable in
    let gconstraints = collectStatementList agstatements
    in
    let afunctions = List.map (fun func -> annotateFunction
     ↪  globalSymbolTable func) functions
    in
    let fconstraints = collectFunctionList afunctions
    in
    let constraints = gconstraints @ fconstraints
```

```
    in
    let unifiedConstraints = unify constraints
    in
    let resolvedGStatements = applyStatementList unifiedConstraints agstatements
    in
    let resolvedFunctions = applyFunctions unifiedConstraints afunctions
    (*in
    let _ = print_string(Ast.string_of_program(resolvedGStatements, resolvedFunctions)) *)
    in
    resolvedGStatements, resolvedFunctions
```

8.5 library.matcv

```
function copyMat(srcMat, destMat)
{
    nDims = <srcMat, 0>;
    matrixSize = 1;

    for (i = 1; i <= nDims; i = i + 1)
    {
        matrixSize = matrixSize * <srcMat, i>;
    }

    matrixSize = matrixSize + 1 + nDims;

    for (i = 0; i < matrixSize; i = i + 1)
    {
        [[destMat, i, <srcMat, i>]];
    }

    srcMat[0];
    return destMat[0];
}

function addMat(destMat, srcMat)
{
    nDims = <srcMat, 0>;
    matrixSize = 1;

    for (i = 1; i <= nDims; i = i + 1)
    {
```

```
        matrixSize = matrixSize * <srcMat, i>;
    }

    matrixSize = matrixSize + 1 + nDims;


    for (i = nDims + 1; i < matrixSize; i = i + 1)
    {
        temp1 = <destMat, i>;
        temp2 = <srcMat, i>;
        [[destMat, i, temp1 + temp2]];
    }

    srcMat[0];
    destMat[0];
    return 0;
}

function minusMat(destMat, srcMat)
{
    nDims = <srcMat, 0>;
    matrixSize = 1;

    for (i = 1; i <= nDims; i = i + 1)
    {
        matrixSize = matrixSize * <srcMat, i>;
    }

    matrixSize = matrixSize + 1 + nDims;

    for (i = nDims + 1; i < matrixSize; i = i + 1)
    {
        temp1 = <destMat, i>;
        temp2 = <srcMat, i>;
        [[destMat, i, temp1 - temp2]];
    }

    srcMat[0];
    destMat[0];
    return 0;
}
```

8.6 codegen.ml

```
(*
 * Code generation for MatCV
*)

module L = Llvm
module A = Ast

let printError message =
    print_string("\nError: " ^ message); exit 1


let printWarning message =
    print_string ("\nWarning: " ^ message)



let mergeSymbolTables globalSymbolTable localSymbolTable = let mergedSymbolTable = Hashtbl.create 100 in
        let _ = Hashtbl.iter (fun key value -> Hashtbl.add mergedSymbolTable key value) localSymbolTable
        in
        let _ = Hashtbl.iter (fun key value -> if not (Hashtbl.mem mergedSymbolTable key) then Hashtbl.add mergedSymbolTable key value) globalSymbolTable
        in mergedSymbolTable



let translate (gstatements, functions) =
  let context = L.global_context () in
  let the_module = L.create_module context "MatCV"
  and i32_t  = L.i32_type  context
  and i8_t   = L.i8_type   context
  and i1_t   = L.i1_type   context
  and void_t = L.void_type context
  and mat_t = L.pointer_type (L.i32_type context) in

  let ltypeOfType = function
      A.Int -> i32_t
    | A.Bool -> i1_t
    | A.Mat(_) -> mat_t
    | A.Void -> void_t
```

```
    | A.Annotation _ -> printError "Unable to resolve symbols"
    | A.Func | A.FuncSignature (_, _)| A.Empty| A.Keyword -> printError "Error during compilation. Should have caught this in semantic check."
  in

  let initOfType = function
    | A.Mat(_) -> (L.const_pointer_null mat_t)
    | t -> L.const_int (ltypeOfType t) 0


  in


  let globalSymbolTable = Hashtbl.create 100
  in
  let mainaexpr = A.ALiteral(0, A.Int)
  in let mainreturnstatement = A.AReturn(A.Int, mainaexpr, A.Void)
  in let gstatements = gstatements @ [(mainreturnstatement)]
  in
  (* Generate code for global statements *)
  let functions = ({A.afname = ("main", A.Func); A.aformals = [] ; A.abody = gstatements; A.retType = A.Int}) :: functions
  in

  let createGlobalVar id idType symbolTable =  let _ = if not (Hashtbl.mem symbolTable id) then
                let init = initOfType idType
                in
                Hashtbl.add symbolTable id ((L.define_global id init the_module), idType)  in ()


  in

  let declareGlobalVariableUsingStatement symbolTable = function
      | A.AVarDecl(x, _) -> (match x with

              | A.AMatrix(id, idType, _, _, _, _) -> createGlobalVar id idType symbolTable
              | A.AExprAssign(id, idType, _, _) -> createGlobalVar id idType symbolTable
              | A.ADimAssign(id, idType, _, _, _) -> createGlobalVar id idType symbolTable
```

```
                | _ -> ()
        )

        | _ -> ()

    in

    (* Generate code for declaration of global variables *)
    let _ = List.iter (declareGlobalVariableUsingStatement globalSymbolTable) gstatements

    in

    let functionTable = Hashtbl.create 100
    in

    let functionDecl afunc =
        let name = (fst afunc.A.afname)
        and formals = Array.of_list (List.map (fun (_, typ) -> ltypeOfType typ) afunc.A.aformals)
        in let ftype = L.function_type (ltypeOfType afunc.A.retType) formals in
        Hashtbl.add functionTable name (L.define_function name ftype the_module, afunc)
        in
        let _ = List.iter functionDecl functions
    in


    (* Generate code for forward declaration of functions *)

    (* TODO: Uncomment the following lines *)

    let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
    let printf_func = L.declare_function "printf" printf_t the_module

    in
    let copyMat_t = L.function_type i32_t [| L.pointer_type i32_t; L.pointer_type i32_t|] in
    let copyMat_func = L.declare_function "copyMat" copyMat_t the_module
    in

```

```
let minusMat_t = L.function_type i32_t [| L.pointer_type i32_t; L.pointer_type i32_t|] in
let minusMat_func = L.declare_function "minusMat" minusMat_t the_module
in

let addMat_t = L.function_type i32_t [| L.pointer_type i32_t; L.pointer_type i32_t|] in
let addMat_func = L.declare_function "addMat" addMat_t the_module
in

let generateFunctionBody afunc =
  let (theFunction, _) = Hashtbl.find functionTable (fst afunc.A.afname)
  in
  let builder = L.builder_at_end context (L.entry_block theFunction)
  in
  let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder

  in
  let memoryMap = Hashtbl.create 100
  in
  let localSymbolTable = Hashtbl.create 100
  in

  let declareFormals (name, typ) value =
      let _ = (L.set_value_name name value;
      let local = L.build_alloca (ltypeOfType typ) name builder
      in
      ignore (L.build_store value local builder);
      Hashtbl.add localSymbolTable name (local, typ)
      )
      in
      ()
  in
  let _ = List.iter2 declareFormals afunc.A.aformals (Array.to_list (L.params theFunction))
  in
  let lookup globalSymbolTable localSymbolTable name = if Hashtbl.mem localSymbolTable name then (fst (Hashtbl.find localSymbolTable name))
                    else (fst (Hashtbl.find globalSymbolTable name))
  in

```

```


  let rec genCodeForExpression globalSymbolTable localSymbolTable memoryMap builder = function
    | A.ALiteral(value, _) -> L.const_int i32_t value
    | A.ABoolLit(value, _) -> L.const_int i1_t (if value then 1 else 0)
    | A.AId(id, _) -> L.build_load (lookup globalSymbolTable localSymbolTable id) id builder
    | A.AUnboundedAccessRead(id, idType, aexpr, _) -> let loadedLMatrix = L.build_load (lookup globalSymbolTable localSymbolTable id) id builder
    in let e = genCodeForExpression globalSymbolTable localSymbolTable memoryMap builder aexpr
    in
    let iPtr = (L.build_in_bounds_gep loadedLMatrix [|e|] (id ^ "_iPtr") builder)
    in
    L.build_load iPtr id builder

    | A.AUnboundedAccessWrite(id, idType, aexpr1, aexpr2, _) -> let loadedLMatrix = L.build_load (lookup globalSymbolTable localSymbolTable id) id builder
    in let e1 = genCodeForExpression globalSymbolTable localSymbolTable memoryMap builder aexpr1
    in let e2 = genCodeForExpression globalSymbolTable localSymbolTable memoryMap builder aexpr2
    in
    let iPtr = (L.build_in_bounds_gep loadedLMatrix [|e1|] (id ^ "_iPtr") builder)
    in
    let _ = L.build_store e2 iPtr builder
    in
    L.build_load iPtr id builder

   | A.AMatPlus(id1, idType1, id2, idType2, _) -> let loadedLMatrix1 = L.build_load (lookup globalSymbolTable localSymbolTable id1) id1 builder
   in let loadedLMatrix2 = L.build_load (lookup globalSymbolTable localSymbolTable id2) id2 builder
   in
   let _ = L.build_call addMat_func [|loadedLMatrix1; loadedLMatrix2|]
   "addMat" builder
```

```
        in
        loadedLMatrix1

        | A.AMatMinus(id1, idType1, id2, idType2, _) -> let loadedLMatrix1 = L.build_load (lookup globalSymbolTable localSymbolTable id1) id1 builder
        in let loadedLMatrix2 = L.build_load (lookup globalSymbolTable localSymbolTable id2) id2 builder
        in
        let _ = L.build_call minusMat_func [|loadedLMatrix1; loadedLMatrix2|]
        "minusMat" builder
        in
        loadedLMatrix1



         (*| A.Size(id, _, _) ->*)
         | A.ANoexpr(_) -> L.const_int i32_t 0
         | A.ABinaryOp(aexpr1, op, aexpr2, _) ->
                let e1 = genCodeForExpression globalSymbolTable localSymbolTable memoryMap builder aexpr1 in let e2 = genCodeForExpression globalSymbolTable localSymbolTable memoryMap builder aexpr2
                in
                (match op with
                        | A.Add     -> L.build_add
                        | A.Sub     -> L.build_sub
                        | A.Mul     -> L.build_mul
                        | A.Div     -> L.build_sdiv
                        | A.And     -> L.build_and
                        | A.Or      -> L.build_or
                        | A.Equal   -> L.build_icmp L.Icmp.Eq
                        | A.Neq     -> L.build_icmp L.Icmp.Ne
                        | A.Less    -> L.build_icmp L.Icmp.Slt
                        | A.Leq     -> L.build_icmp L.Icmp.Sle
                        | A.Greater -> L.build_icmp L.Icmp.Sgt
                        | A.Geq     -> L.build_icmp L.Icmp.Sge
                        | A.Mod     -> L.build_srem
                        (* TODO: Change the code for Exp
                         * Following code is just a placeholder
                         *)
                        | A.Exp     -> L.build_mul
                 ) e1 e2 "tmp" builder
```

```
    | A.AUnop(uop, aexpr, exprType) ->
            let e = genCodeForExpression globalSymbolTable localSymbolTable memoryMap builder aexpr in
            (match uop with
            | A.Neg -> L.build_neg
            | A.Not -> L.build_not) e "tmp" builder
    | A.ACall ("print", _, [aexpr], _) | A.ACall ("printb", _, [aexpr], _) ->
    L.build_call printf_func [| int_format_str ; (genCodeForExpression globalSymbolTable localSymbolTable memoryMap builder aexpr)|]
      "printf" builder
    | A.ACall(id, _, aExprList, _) -> let actuals = List.map (fun aexpr -> genCodeForExpression globalSymbolTable localSymbolTable memoryMap builder aexpr) aExprList
                                  in
                                  let (lfunc, afunc) = Hashtbl.find functionTable id
                                  in
                                  let retType = afunc.A.retType
                                  in
                                  let result = (match retType with
                                                  | A.Void -> ""
                                                  | _ -> id ^ "_result") in
                                      L.build_call lfunc (Array.of_list actuals) result builder
  | A.AMatAccess(id, idType, aExprList, nDims, exprType) ->
              let dimIndices = List.map (fun aexpr -> genCodeForExpression globalSymbolTable localSymbolTable memoryMap builder aexpr) aExprList
              in
              let lMatrix = (lookup globalSymbolTable localSymbolTable id)
              in
              let loadedLMatrix = L.build_load lMatrix (id ^ "_load") builder
              in
              let rec findDimSizes dimSizeList = function
```

```
                    | 0 -> dimSizeList
                    | i -> findDimSizes (
                        (
                            let dimPtr = (L.build_in_bounds_gep loadedLMatrix [|L.const_int i32_t i|] (id ^ "_dim_" ^ string_of_int i) builder)
                            in
                            L.build_load dimPtr (id ^ "_dim_" ^ string_of_int i ^ "value_") builder

                        )
                        ::dimSizeList) (i - 1)

            in let dimSizes = findDimSizes [] nDims
            in
            let (index, _) = List.fold_left2 (fun (result, productDim) size_i index_i ->
                    let productDimMulIndex = (L.build_mul productDim index_i ("tmp_") builder)
                    in let result = (L.build_add result productDimMulIndex ("tmp2_") builder)
                    in
                    let productDim = (L.build_mul productDim size_i ("tmp3_") builder)
                    in
                    (result, productDim) )
           ((L.const_int i32_t (1 + nDims)), (L.const_int i32_t 1))
           (List.rev dimSizes) (List.rev dimIndices)
            in
            let elementPtr = L.build_in_bounds_gep loadedLMatrix [|index|] (id ^ "_element") builder
            in
            L.build_load elementPtr (id ^ "_element") builder


in


let freeMatrixFun lMatrix id builder =
      (* FREE: *)
      let freeMatrix = L.build_load lMatrix (id ^ "_free") builder
      in
      let _ = L.build_free freeMatrix builder
```

```
      in
      ()

in


let genCodeForVarDecl globalSymbolTable localSymbolTable memoryMap builder = function
  | A.ANodecl(_) -> let _ = L.const_int i32_t 0 in ()
  | A.AMatrix(id, idType, aExprListList, nRows, nCols, _) ->
          let _ = (if Hashtbl.mem localSymbolTable id then
                      if Hashtbl.mem memoryMap id then
                          (* free, malloc *)
                          (* FREE: *)
                          let _ = freeMatrixFun  (lookup globalSymbolTable localSymbolTable id) id builder
                          in
                          (* MALLOC *)
                          let mallocMatrix = L.build_array_malloc i32_t (L.const_int i32_t (nRows * nCols + 3)) (id ^ "_malloc") builder
                          in
                          let _ = L.build_store mallocMatrix (lookup globalSymbolTable localSymbolTable id) builder
                          in ()
                      else
                          (* malloc, add to memory map *)
                          (* MALLOC *)
                          let mallocMatrix = L.build_array_malloc i32_t (L.const_int i32_t (nRows * nCols + 3)) (id ^ "_malloc") builder
                          in
                          let _ = L.build_store mallocMatrix (lookup globalSymbolTable localSymbolTable id) builder
                          in
                          (* Add to memory map *)
                          Hashtbl.add memoryMap id (lookup globalSymbolTable localSymbolTable id)

                  else if Hashtbl.mem globalSymbolTable id then
```

```
                (* free, malloc *)
                (* FREE: *)
                let _ = freeMatrixFun  (lookup globalSymbolTable localSymbolTable id) id builder
                in
                (* MALLOC *)
                let mallocMatrix = L.build_array_malloc i32_t (L.const_int i32_t (nRows * nCols + 3)) (id ^ "_malloc") builder
                in
                let _ = L.build_store mallocMatrix (lookup globalSymbolTable localSymbolTable id) builder
                in
                ()
        else
            (* alloc ptr, malloc, add to local map, add to memory map *)
            (* Alloc *)
            let lMatrix = L.build_alloca mat_t id builder
            in
            let _ = Hashtbl.add localSymbolTable id (lMatrix, idType)
            in
            let _ = Hashtbl.add memoryMap id lMatrix
            in
            (* MALLOC *)
            let mallocMatrix = L.build_array_malloc i32_t (L.const_int i32_t (nRows * nCols + 3)) (id ^ "_malloc") builder
            in
            let _ = L.build_store mallocMatrix lMatrix builder
            in
            ()
) in
(* Assign dimensions and values *)
  let lMatrix = (lookup globalSymbolTable localSymbolTable id)
  in
  let loadedLMatrix = L.build_load lMatrix (id ^ "_load") builder
  in
```

```
                let zeroIndexPtr = L.build_in_bounds_gep loadedLMatrix
                 ↪  [|L.const_int i32_t 0|] (id ^ "_zero_index")
                 ↪  builder
                in
                let _ = L.build_store (L.const_int i32_t 2)
                 ↪  zeroIndexPtr builder
                in
                let oneIndexPtr = L.build_in_bounds_gep loadedLMatrix
                 ↪  [|L.const_int i32_t 1|] (id ^ "_one_index") builder
                in
                let _ = L.build_store (L.const_int i32_t nRows)
                 ↪  oneIndexPtr builder
                in
                let secondIndexPtr = L.build_in_bounds_gep
                 ↪  loadedLMatrix [|L.const_int i32_t 2|] (id ^
                 ↪  "_two_index") builder
                in
                let _ = L.build_store (L.const_int i32_t nCols)
                 ↪  secondIndexPtr builder
                in
                let _ = List.fold_left (fun acc aExprList ->
          List.fold_left (fun acc aexpr -> let lvalue =
           ↪  genCodeForExpression globalSymbolTable localSymbolTable
           ↪  memoryMap builder aexpr in
          let ptrIndex = L.build_in_bounds_gep loadedLMatrix
           ↪  [|L.const_int i32_t acc|] (id ^ "_ptr_index") builder in
          let _ = L.build_store lvalue ptrIndex builder in
                        (acc + 1)) acc aExprList
            ) 3 aExprListList
              in
              ()

    | A.ADimAssign(id, idType, aExprList, nDims,  _) ->
                            let (productSizes, dimSizeList) =
                             ↪  List.fold_left (fun (acc, dimSizeLst)
                             ↪  aexpr -> let laexpr =
                             ↪  genCodeForExpression globalSymbolTable
                             ↪  localSymbolTable memoryMap builder
                             ↪  aexpr in let dimLst =
                                dimSizeLst @ [laexpr]
                            in
                            (L.build_mul acc laexpr "expr_prod_size"
                             ↪  builder, dimLst)) (L.const_int i32_t 1,
                             ↪  []) aExprList
```

```
        in
        let matrixSize = (L.build_add productSizes  (L.const_int i32_t (nDims + 1)) "mat_size" builder)
        in
                let _ = (if Hashtbl.mem localSymbolTable id then
                        if Hashtbl.mem memoryMap id then
                            (* free, malloc *)
                            (* FREE: *)
                            let _ = freeMatrixFun  (lookup globalSymbolTable localSymbolTable id) id builder
                            in
                            (* MALLOC *)
                            let mallocMatrix = L.build_array_malloc i32_t matrixSize (id ^ "_malloc") builder
                            in
                            let _ = L.build_store mallocMatrix (lookup globalSymbolTable localSymbolTable id) builder
                            in ()
                        else
                            (* malloc, add to memory map *)
                            (* MALLOC *)
                            let mallocMatrix = L.build_array_malloc i32_t matrixSize (id ^ "_malloc") builder
                            in
                            let _ = L.build_store mallocMatrix (lookup globalSymbolTable localSymbolTable id) builder
                            in
                            (* Add to memory map *)
                            Hashtbl.add memoryMap id (lookup globalSymbolTable localSymbolTable id)

                    else if Hashtbl.mem globalSymbolTable id then
                            (* free, malloc *)
                            (* FREE: *)
                            let _ = freeMatrixFun  (lookup globalSymbolTable localSymbolTable id) id builder
                            in
```

```
                (* MALLOC *)
                let mallocMatrix = L.build_array_malloc i32_t matrixSize (id ^ "_malloc") builder
                in
                let _ = L.build_store mallocMatrix (lookup globalSymbolTable localSymbolTable id) builder
                in
                ()
        else
            (* alloc ptr, malloc, add to local map, add to memory map *)
            (* Alloc *)
            let lMatrix = L.build_alloca mat_t id builder
            in
            let _ = Hashtbl.add localSymbolTable id (lMatrix, idType)
            in
            let _ = Hashtbl.add memoryMap id lMatrix
            in
            (* MALLOC *)
            let mallocMatrix = L.build_array_malloc i32_t matrixSize (id ^ "_malloc") builder
            in
            let _ = L.build_store mallocMatrix lMatrix builder
            in
            ()
) in
  let lMatrix = (lookup globalSymbolTable localSymbolTable id)
  in
  let loadedLMatrix = L.build_load lMatrix (id ^ "_load") builder
  in
  let zeroIndexPtr = L.build_in_bounds_gep loadedLMatrix [|L.const_int i32_t 0|] (id ^ "_zero_index") builder
  in
  let _ = L.build_store (matrixSize) zeroIndexPtr builder
  in
  let _ = List.fold_left (fun acc lvalue ->
```

```
            let ptrIndex = L.build_in_bounds_gep loadedLMatrix
             ↪  [|L.const_int i32_t acc|] (id ^ "_ptr_index") builder in
            let _ = L.build_store lvalue ptrIndex builder in
                            (acc + 1)) 1 dimSizeList
                  in
                ()

    | A.AMatElementAssign(id, idType, aExprList, aExpr, nDims, _) ->
                  let dimIndices = List.map (fun aexpr ->
                   ↪  genCodeForExpression globalSymbolTable
                   ↪  localSymbolTable memoryMap builder aexpr) aExprList
                  in
                  let lMatrix = (lookup globalSymbolTable
                   ↪  localSymbolTable id)
                  in
                  let loadedLMatrix = L.build_load lMatrix (id ^ "_load")
                   ↪  builder
                  in
                  let rec findDimSizes dimSizeList = function
                          | 0 -> dimSizeList
                          | i -> findDimSizes (
                              (
                                  let dimPtr = (L.build_in_bounds_gep
                                   ↪  loadedLMatrix [|L.const_int i32_t
                                   ↪  i|] (id ^ "_dim_" ^ string_of_int
                                   ↪  i) builder)
                                  in
                                  L.build_load dimPtr (id ^ "_dim_" ^
                                   ↪  string_of_int i ^ "value_") builder

                              )
                              ::dimSizeList) (i - 1)

                  in let dimSizes = findDimSizes [] nDims
                  in
                  let (index, _) = List.fold_left2 (fun (result,
                   ↪  productDim) size_i index_i ->
                          let productDimMulIndex = (L.build_mul
                           ↪  productDim index_i ("tmp_") builder)
                          in let result = (L.build_add result
                           ↪  productDimMulIndex ("tmp2_") builder)
                          in
                          let productDim = (L.build_mul productDim size_i
                           ↪  ("tmp3_") builder)
```

```
                        in
                        (result, productDim) )
                    ((L.const_int i32_t (1 + nDims)), (L.const_int i32_t 1))
                    (List.rev dimSizes) (List.rev dimIndices)
                    in
                    let elementPtr = L.build_in_bounds_gep loadedLMatrix [|index|] (id ^ "_element") builder
                    in
                    let eValue = genCodeForExpression globalSymbolTable localSymbolTable memoryMap builder aExpr in
                    let _ = L.build_store eValue elementPtr builder in ()


    | A.AExprAssign(id, idType, aExpr, _) ->
            (
            match idType with
            | A.Mat(nDims) ->
                let loadedLMatrix = genCodeForExpression globalSymbolTable localSymbolTable memoryMap builder aExpr
                in
                let rec findDimSizes dimSizeList = function
                        | 0 -> dimSizeList
                        | i -> findDimSizes (
                            (
                                let dimPtr = (L.build_in_bounds_gep loadedLMatrix [|L.const_int i32_t i|] (id ^ "_dim_" ^ string_of_int i) builder)
                                in
                                L.build_load dimPtr (id ^ "_dim_" ^ string_of_int i ^ "value_") builder

                            )
                            ::dimSizeList) (i - 1)

                in let dimSizes = findDimSizes [] nDims
                in
                      let productSizes  = List.fold_left (fun acc laexpr ->
                      (L.build_mul acc laexpr "expr_prod_size" builder)) (L.const_int i32_t 1) dimSizes
                      in
```

```
                let matrixSize = (L.build_add productSizes
                 ↪  (L.const_int i32_t (nDims + 1)) "mat_size"
                 ↪  builder)
            in

            let _ = (if Hashtbl.mem localSymbolTable id then
                    if Hashtbl.mem memoryMap id then
                        (* free, malloc *)
                        (* FREE: *)
                        let _ = freeMatrixFun  (lookup
                         ↪  globalSymbolTable localSymbolTable
                         ↪  id) id builder
                        in
                        (* MALLOC *)
                        let mallocMatrix = L.build_array_malloc
                         ↪  i32_t matrixSize (id ^ "_malloc")
                         ↪  builder
                        in
                        let _ = L.build_store mallocMatrix
                         ↪  (lookup globalSymbolTable
                         ↪  localSymbolTable id) builder
                        in ()
                    else
                        (* malloc, add to memory map *)
                        (* MALLOC *)
                        let mallocMatrix = L.build_array_malloc
                         ↪  i32_t matrixSize (id ^ "_malloc")
                         ↪  builder
                        in
                        let _ = L.build_store mallocMatrix
                         ↪  (lookup globalSymbolTable
                         ↪  localSymbolTable id) builder
                        in
                        (* Add to memory map *)
                        Hashtbl.add memoryMap id (lookup
                         ↪  globalSymbolTable localSymbolTable
                         ↪  id)

                else if Hashtbl.mem globalSymbolTable id then
                        (* free, malloc *)
                        (* FREE: *)
                        let _ = freeMatrixFun  (lookup
                         ↪  globalSymbolTable localSymbolTable
                         ↪  id) id builder
```

```
                            in
                            (* MALLOC *)
                            let mallocMatrix = L.build_array_malloc
                             ↪  i32_t matrixSize (id ^ "_malloc")
                             ↪  builder
                            in
                            let _ = L.build_store mallocMatrix
                             ↪  (lookup globalSymbolTable
                             ↪  localSymbolTable id) builder
                            in
                            ()
                    else
                        (* alloc ptr, malloc, add to local map, add
                         ↪  to memory map *)
                        (* Alloc *)
                        let lMatrix = L.build_alloca mat_t id builder
                        in
                        let _ = Hashtbl.add localSymbolTable id
                         ↪  (lMatrix, idType)
                        in
                        let _ = Hashtbl.add memoryMap id lMatrix
                        in
                        (* MALLOC *)
                        let mallocMatrix = L.build_array_malloc i32_t
                         ↪  matrixSize (id ^ "_malloc") builder
                        in
                        let _ = L.build_store mallocMatrix lMatrix
                         ↪  builder
                        in
                        ()
                ) in
                  let lMatrix = (lookup globalSymbolTable
                   ↪  localSymbolTable id)
                  in
                  let loadedAllocMat = L.build_load lMatrix (id ^
                   ↪  "_load") builder
                  in
                  let _ = L.build_call copyMat_func [|loadedLMatrix;
                   ↪  loadedAllocMat|]
                  "copyMat" builder
                  in
                  ()
```

```
            | 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 ()
                else
                    let local = L.build_alloca (ltypeOfType typ) id
                     ↪  builder
                    in
                    ignore (L.build_store (genCodeForExpression
                     ↪  globalSymbolTable localSymbolTable memoryMap
                     ↪  builder aExpr) local builder);
                    Hashtbl.add localSymbolTable id (local, typ)
            )

    in

    let addTerminal builder f =
      (match L.block_terminator (L.insertion_block builder) with
        Some _ -> ()
      | None -> ignore (f builder))
    in

    let rec genCodeForStatements ?contBB:(contBB = None)
     ↪  globalSymbolTable localSymbolTable memoryMap builder = function
        | A.ABlock(astatementList, _) ->  let newGlobalSymbolTable =
         ↪  mergeSymbolTables globalSymbolTable localSymbolTable in
                                  let newLocalSymbolTable = Hashtbl.create
                                   ↪  100 in
                                  let newMemoryMap = Hashtbl.create 100 in
                                  List.fold_left (fun builder astatement
                                   ↪  -> genCodeForStatements
                                   ↪  ~contBB:contBB newGlobalSymbolTable
                                   ↪  newLocalSymbolTable newMemoryMap
                                   ↪  builder astatement) builder
                                   ↪  astatementList
        | A.AExpr(aExpr, _) -> let _ = genCodeForExpression
         ↪  globalSymbolTable localSymbolTable memoryMap builder aExpr
         ↪  in builder

        | A.AVarDecl(aVarDecl, _) -> let _ = genCodeForVarDecl
         ↪  globalSymbolTable localSymbolTable memoryMap builder
         ↪  aVarDecl
```

```
                    in builder

        | A.AReturn(retType, aExpr, _) -> ignore (match afunc.A.retType with
                                        | A.Void -> L.build_ret_void builder
                                        | _ -> L.build_ret (genCodeForExpression globalSymbolTable localSymbolTable memoryMap builder aExpr)
                                                  builder); builder
      | A.AIf (predicate, thenStatement, elseStatement, _) ->
              let boolVal = genCodeForExpression globalSymbolTable localSymbolTable memoryMap builder predicate in
              let mergeBB = L.append_block context "merge" theFunction in

              let thenBb = L.append_block context "then" theFunction in
              addTerminal (genCodeForStatements ~contBB:contBB globalSymbolTable localSymbolTable memoryMap (L.builder_at_end context thenBb) thenStatement)
                (L.build_br mergeBB);

              let elseBb = L.append_block context "else" theFunction in
              addTerminal (genCodeForStatements ~contBB:contBB globalSymbolTable localSymbolTable memoryMap (L.builder_at_end context elseBb) elseStatement)
                (L.build_br mergeBB);

              ignore (L.build_cond_br boolVal thenBb elseBb builder);
              L.builder_at_end context mergeBB

      | A.AWhile (predicate, body, _) ->
               let predBB = L.append_block context "while" theFunction in
               ignore (L.build_br predBB builder);
```

```

                    let bodyBB = L.append_block context "while_body"
                     ↪  theFunction in
                    addTerminal (genCodeForStatements globalSymbolTable
                     ↪  localSymbolTable memoryMap ~contBB:(Some predBB)
                     ↪  (L.builder_at_end context bodyBB) body)
                      (L.build_br predBB);

                    let predBuilder = L.builder_at_end context predBB in
                    let boolVal = genCodeForExpression globalSymbolTable
                     ↪  localSymbolTable memoryMap predBuilder predicate
                     ↪  in

                    let mergeBB = L.append_block context "merge"
                     ↪  theFunction in
                    ignore (L.build_cond_br boolVal bodyBB mergeBB
                     ↪  predBuilder);
                    L.builder_at_end context mergeBB



        | A.AFor(aVarDecl1, aExpr, aVarDecl2, aStatement, _) ->
                  (let newGlobalSymbolTable = mergeSymbolTables
                   ↪  globalSymbolTable localSymbolTable in
                  let newLocalSymbolTable = Hashtbl.create 100 in
                  let newMemoryMap = Hashtbl.create 100 in
                  let builder = genCodeForStatements ~contBB:contBB
                   ↪  newGlobalSymbolTable newLocalSymbolTable
                   ↪  newMemoryMap builder (A.AVarDecl(aVarDecl1,
                   ↪  A.Void))
                  in
                  let incrBB = L.append_block context "forincr"
                   ↪  theFunction in
                  let _ = (genCodeForStatements ~contBB:contBB
                   ↪  newGlobalSymbolTable newLocalSymbolTable
                   ↪  newMemoryMap (L.builder_at_end context incrBB)
                   ↪  (A.AVarDecl(aVarDecl2, A.Void)))
                  in
                  let forBB = L.append_block context "for" theFunction in
                  let _ = L.build_br forBB builder in
                  let _ = L.build_br forBB (L.builder_at_end context
                   ↪  incrBB) in
                  let predBuilder = L.builder_at_end context forBB in
```

```
                    let boolVal = genCodeForExpression newGlobalSymbolTable newLocalSymbolTable newMemoryMap predBuilder aExpr in
                    let bodyBB = L.append_block context "forbody" theFunction in
                    let _ = addTerminal (genCodeForStatements newGlobalSymbolTable newLocalSymbolTable newMemoryMap ~contBB:(Some incrBB) (L.builder_at_end context bodyBB) aStatement) (L.build_br incrBB)
                    in
                    let mergeBB = L.append_block context "merge" theFunction in
                    ignore (L.build_cond_br boolVal bodyBB mergeBB predBuilder);
                    L.builder_at_end context mergeBB


          | A.AExit(_) -> builder
          | A.ABreak(_) -> builder
          | A.AForEachLoop (_, _, _, _, _, _, _) -> builder
          | A.AContinue(_) -> let _ = (match contBB with
                                  | Some x -> L.build_br x builder
                                  | _ -> printError ("Code should never reach here!")) in builder

        in
        let builder = genCodeForStatements globalSymbolTable localSymbolTable memoryMap builder (A.ABlock (afunc.A.abody, A.Void)) in
                addTerminal builder (match afunc.A.retType with
                                        | A.Void -> L.build_ret_void
                                        | t -> L.build_ret (initOfType t))

  in
  (* Generate code for function definitions *)
  let _ = List.iter generateFunctionBody functions
  in

  the_module
```

8.7 matcv.ml

```
(* Top-level of the MicroC compiler: scan & parse the input,
   check the resulting AST, generate LLVM IR, and dump the module *)
open Ast
open Llvm

type action = Ast | LLVM_IR | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);        (* Print the AST only *)
                              ("-l", LLVM_IR);  (* Generate LLVM, don't check *)
                              ("-c", Compile) ] (* Generate, check LLVM IR *)
  else Compile in
  let lexbuf = Lexing.from_channel stdin in
  let ast = Parser.program Scanner.token lexbuf in
  let sast = Semant.check_semantics ast in
    match action with
    Ast ->
    print_string(Ast.string_of_program(sast))
  | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate sast))
  | Compile -> let m = Codegen.translate sast in
    Llvm_analysis.assert_valid_module m;
    print_string (Llvm.string_of_llmodule m)
```

8.8 Makefile

```
# Make sure ocamlbuild can find opam-managed packages: first run
#
# eval `opam config env`

# Easiest way to build: using ocamlbuild, which in turn uses ocamlfind

.PHONY : matcv.native

matcv.native :
	ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis \
		matcv.native

```

```
# "make clean" removes all generated files

.PHONY : clean
clean :
	ocamlbuild -clean
	rm -rf testall.log *.diff matcv scanner.ml parser.ml parser.mli
	rm -rf *.cmx *.cmi *.cmo *.cmx *.o

# More detailed: build using ocamlc/ocamlopt + ocamlfind to locate LLVM

OBJS = ast.cmx codegen.cmx parser.cmx scanner.cmx semant.cmx matcv.cmx

matcv : $(OBJS)
	ocamlfind ocamlopt -linkpkg -package llvm -package llvm.analysis $(OBJS) -o matcv

scanner.ml : scanner.mll
	ocamllex scanner.mll

parser.ml parser.mli : parser.mly
	ocamlyacc parser.mly

%.cmo : %.ml
	ocamlc -c $<

%.cmi : %.mli
	ocamlc -c $<

%.cmx : %.ml
	ocamlfind ocamlopt -c -package llvm $<

### Generated by "ocamldep *.ml *.mli" after building scanner.ml and parser.ml
ast.cmo :
ast.cmx :
codegen.cmo : ast.cmo
codegen.cmx : ast.cmx
matcv.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo
matcv.cmx : semant.cmx scanner.cmx parser.cmx codegen.cmx ast.cmx
parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmi
scanner.cmo : parser.cmi
```

```
scanner.cmx : parser.cmx
semant.cmo : ast.cmo
semant.cmx : ast.cmx
parser.cmi : ast.cmo

# Building the tarball

TESTS = add1 arith1 arith2 arith3 fib for1 for2 func1 func2 func3        \
    func4 func5 func6 func7 func8 gcd2 gcd global1 global2 global3        \
    hello if1 if2 if3 if4 if5 local1 local2 ops1 ops2 var1 var2                 \
    while1 while2

FAILS = assign1 assign2 assign3 dead1 dead2 expr1 expr2 for1 for2        \
    for3 for4 for5 func1 func2 func3 func4 func5 func6 func7 func8        \
    func9 global1 global2 if1 if2 if3 nomain return1 return2 while1        \
    while2

TESTFILES = $(TESTS:%=test-%.mc) $(TESTS:%=test-%.out) \
            $(FAILS:%=fail-%.mc) $(FAILS:%=fail-%.err)

TARFILES = ast.ml codegen.ml Makefile matcv.ml parser.mly README scanner.mll \
        semant.ml testall.sh $(TESTFILES:%=tests/%)

matcv-llvm.tar.gz : $(TARFILES)
        cd .. && tar czf matcv-llvm/matcv-llvm.tar.gz \
              $(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