C?

Andrew Aday, Amol Kapoor, Jonathan Zhang
(aza2112, ajk2227, jz2814)@columbia.edu
https://github.com/AndrewAday/CQM

December 2017

Contents

1 Abstract

In this document we propose C?, a multi-paradigm imperative general purpose language. C? is designed to provide a foundation for domain specific applications through the development of powerful yet simple libraries. Input to the C? compiler mimics the familiar C coding syntax. The compiler outputs an C? executable, using LLVM as an intermediate state. We demonstrate the flexibility of C? by providing and explaining a simple Deep Learning library, C? DEEP. This module contains the essential components that allow users to quickly and flexibly develop neural networks models in C?.

2 Introduction

2.1 Overview

High level programming languages like Python enjoy widespread use because of powerful community-built libraries that make domain specific applications easier. For example, the Numpy and Pandas libraries have made Python essential for most data science applications. Similarly, the Tensorflow and Theano libraries have made Python the de-facto language of Machine Learning. Rails almost single-handedly made Ruby relevant again. And of course, though Javascript is a mess, it continues to be a popular language with dozens of useful libraries. Prolog, on the other hand, sees very little use outside of niche cases - the difficulty of library development in Prolog results in fewer libraries and dwindling interests in the language. From these and other historical examples, it is clear that the development of a good[1] language depends less on its up-front domain application and more on the ability to quickly abstract the language for other uses. Library generation in turn depends on the ease of use of the syntax and the features available in the base language.

2.2 Syntax

Many popular programming languages, such as Python and Javascript, are syntactically fluid. Generally, such languages share the following syntactic features: types are clear and accessible or they do not matter, syntax is human readable, and the language is forgiving. Python, for example, is often described as a language that 'just works'. While these are admirable traits for scripting, library development is made significantly harder by a forgiving language. In order to sustain a large number of use cases, libraries need to be bug free and extremely tolerant to poorly written user code. Fluid syntax means that a) there are more avenues for a user to misuse a library (and therefore more edge cases a library developer needs to check); and b) without a strict syntax and compiler, it is harder to write fault-tolerant due to the universality of human error. While the

[1] Or at least, widely used.

trade off of between easy writing and easy debugging has no obvious right answer in the world of language design, we feel strongly that library development is specifically well suited to a strongly typed language.

2.3 Features

Based on previous research[2], we concluded that the following features are vital in modern programming contexts to quickly develop powerful domain specific libraries.

- **Structs/User Defined Types.** Many libraries exist to provide frameworks that allow users to manipulate domain-specific data types. Including structs allow library developers to extend the type system of a language to include user-defined constructs. For example, our deep learning library[3] provides a $\texttt{fc}_{m}odelstruct for defining the layers that compose a deep learning model. By providing a layer of a forward neural network applications.$

- **Function Pointers.** Libraries often rely on the ability to pass around functions as types in order to implement features like callback options for asynchronous behavior or primitive polymorphism when combined with structs. Function pointers allow functions to be referenced without directly calling them, allowing a user defined function to be passed in at run time by the user or library developer. Importantly, function pointers also allow library developers to leave placeholders where a user is expected to pass their custom functions. Taking our neural networks example from above, users may want to apply different types of cost functions to their network, which necessitates the user of function pointers to these custom procedures.

- **External Linking/Matrices.** A programming language is expected to be efficient. Due to the difficulty in beating the speed of C, many modern languages are either directly implemented on top of C or provide easy ways to run external source code. For example, much of Python is written in Cython, a lightweight wrapper over the underlying C code that allows for the speed and usability of Python. Our language will make extensive use of linking with an underlying C++ library to provide matrix functionality and fast math operations. Including this feature inherently into the language opens library developers to a vast array of different data science and mathematical possibilities.

- **Collections of data.** In a data oriented world, more and more end applications of programming rely on crunching numbers. Native arrays, matrices, and fast operation support are therefore a vitally important feature to include in any modern language.

[2] We sat and thought about this for a while.
[3] see section 4.7.5

2.4 C?

The central design goal of C? is to build a flexible language that can be used for many different paradigms and many different domain applications. With this in mind, C? implements the syntax and features described above. C? syntax is C-like syntax that piggybacks on the widespread familiarity of C while maintaining C typing rules. We believe that this is a compromise between scripting and strong typing that will allow a large set of developers already familiar with C to be able to code in C? with minimal study. It will also enforce stricter code generation, saving time for users on debugging. C? implements native structs, function pointers, arrays, and matrices. C? also allows users to link directly to any C functions. We use all of these features in the development and implementation of the C? DEEP deep learning library, which serves as a proof of concept for the general power of C? as a language.

3 C? Language Tutorial: Welcome to C?!

Thank you for downloading C?! The following tutorial is meant for newcomers to the C? language. The tutorial provides basic instructions for getting started with C?, as well as an introduction to more advanced C? constructs.

3.1 Hello World and Compiler Usage

Once you have downloaded the C? zip file, unzip the folder and enter the C? directory. Type **make** to create the C? compiler, which will take C? code and convert it to a machine readable executable. Once the C? compiler has been made, programs can be compiled by running the `compile.sh` shell script.

The following sample code is a simple implementation of Hello World in C?.

```
1 int main() {
2     print_string( "Hello World!" );
3     return 0;
4 }
```

Enter the above code into a file named `hello.cqm`. To compile and run this code, type:

```
> ./compile.sh hello.C?
> ./hello.exe
Hello World!
```

Lets walk through the code line by line.

- C? files need to have a `int main()` function. This is where program execution begins. Functions in C? have return types, meaning the entire block of code will spit out a piece of data of the given type. By default, `main` returns an int type, so we define the function as `int main()`. Because all of the code that follows happens inside `main`, we add an open curly brace to tell the program how the following code relates to the previous code. See more on Functions below.
- `print_string` is a built in function, meaning that it can be called in any C? file without having to define it separately. As one can imagine, this function prints out strings of characters. There are other `print_<type>` functions for various other types, e.g. `print_float, print_mat`. Printing integers is a simple `print`. We pass in a string "Hello World". The string type is indicated by the quotation marks. Note that this line ends with a semi-colon. All statements in C? must end in a semicolon or in a curly brace. Semi-colons are used to tell the compiler when a line ends.

- We indicated at the top that `main` returns a type `int`, i.e. an integer. Since we do not actually care about the type, we simply `return 0;`.
- Finally, we close out with a closing curly brace to tell the program that `main` has ended.

Now lets examine how we ran the code.

- Once we have our `hello.C?` file, we need to convert it to something the computer can read. We use `compile.sh` to take our C? code and turn it into a machine executable.
- Now that we have a machine executable, we can run it like any other executable with the `./` syntax.

And that's it! Congrats on writing your first piece of C? code! Read on for more features.

3.2 Arithmetic, Algebra, and If/Else

3.2.1 Basic Math

Like any programming language, C? supports basic algebraic operations through variable declaration and assignment. This section will serve as an introduction to variables and operation usage in C?. Though we limit our discussion to integers and booleans here, note that many principles can be extended to other types (including more advanced types like structs and function pointers).
Basic math is fairly straight forward:

```
1  int main() {
2      print( 1 + 1 );          /* 2 */
3      print( 2*4 );            /* 8 */
4      print( (3 + 3)*2/3 );    /* 4 */
5      return 0;
6  }
```

Note that above we make use of comments. These are described further in the Style section of the tutorial. For now, just know that comments are ignored by C? code.

3.2.2 Variables

Variables can be declared and used as follows:

```
1  int main() {
2      int x;
3      int y;
```

```
    int z;

    x = 6;
    y = 2;
    z = 3;
    print( x*y/z );     /* 4 */
    return 0;
}
```

Note that all variables must be declared at the top of the function that is using them.

3.2.3 Booleans

Boolean algebra can also be implemented in C?. For example:

```
int main() {
    printb(true);           /* true */
    printb(false);          /* false */
    printb(true && false);  /* false */
    printb(true || false);  /* true */
    return 0;
}
```

Above we make use of the `&&` and `||` operators, which correspond to AND and OR respectively. C? also allows various comparisons that can return boolean types. For example:

```
int main() {
    printb(1 == 2);     /* false */
    printb(1 == 1);     /* true */
    printb(1 < 2);      /* true */
    printb(1 != 2);     /* true */
    return 0;
}
```

Above we make use of the `==`, `!=` and `<` operators, which correspond to EQUALS, NOT EQUALS, and LESS THAN respectively. To see the full list of available boolean operators, consult the Language Reference Manual. Combining various boolean terms allows for complex conditional statements. For example:

```
/* true if x equals y OR x is less than y and y equals z */
printb( ( x == y ) || ((x < y) && y == z) );
```

3.2.4 Conditionals

C? can utilize the powerful boolean expressions above to create branching statements in code that are dependent on certain values during execution. For example:

```
int main() {
    int x;
    int y;

    x = 5;
    y = 1;

    if ( x < y ) {
        print(x);
    } else {
        print(y);   /* 1 */
    }
}
```

The above code utilizes a programming construct known as an if/else block. As the name implies, the program will run the code in the if block *if* the corresponding boolean statement is true. Otherwise, it runs the code in the else block. These switches can be strung together. For example:

```
int main() {
    int x;
    int y;

    x = 5;
    y = 1;

    if ( x < y ) {
        print(x);
    } else {
        if (y != 1) {
            print y;
        } else {
            print 1;    /* 1 */
        }
    }
}
```

If/else blocks provide the first introduction to control flow, i.e. the tools available to indicate how control should flow through various components of the code.

3.3 Collections of Data and Loops

3.3.1 While, For

Many programs require repetitive actions that can be simplified with a looping syntax. In C?, there are two kinds of loops: while loops, and for loops. Both are equivalent, but are optimized for slightly different use cases. They are presented below:

```
int main() {
    int i;

    i = 0;
    while ( i < 10 ) {
        i = i + 1;
        print( i );
    }

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

A while loop is the most basic kind of loop. It will continue to run the code in between the curly braces *while* some value in the parenthesis is true. In the example above, the loop continues *while* the variable value of `i` is less than 10. A for loop is a slightly more complex while loop. Unlike a while loop, which only checks a single boolean condition, a for loop has three components separated by semi-colons. The first component runs before the for loop - above, we use it to set the variable `i` to 0. The second component is the condition, and functions the same way as the boolean in the while loop. The third component runs on each step of the for loop - we use it here to increment the value of `i`. Like if/else blocks, loops are an important tool in the control flow toolbox.

3.3.2 Arrays

It can often be useful to have data stored consecutively. C? allows for typed arrays that can contain sequences of the same type of object. For example:

```
int main() {
    float[] f_arr;
    f_arr = make(float, 0);
    f_arr = append(f_arr, 1.);

    print_float(f_arr[1]);
```

```
    return 0;
}
```

In the above code, we create a float array. Arrays require a special `make` function to create the space where the data will be stored. We can also `append` new values to the array, and then index the array to print out individual values. Other array specific functions can be found in the Language Reference Manual. Arrays can be manipulated with for loops, providing a powerful tool for data manipulation. For example, basic array iteration can be accomplished as follows:

```
int main() {
    int i;
    int[] i_arr;

    i_arr = (int[]) {1, 2, 3, 4};      /* Array literal */

    for (i = 0; i < len(i_arr); i++) {
        print(i_arr[i]);                /* Prints out created array */
    }

    return 0;
}
```

Note that C? does *not* allow arrays of arrays (see the Matrix type for nested array functionality). Further, note that arrays are heap objects (see the Memory Management section below) and are pass by reference.

3.4 Functions

C? allows users to split off blocks of code into reusable functions. Each function requires a type, a unique function name, and zero or more arguments to be used inside the function. Functions can then be 'called' by providing the name of the function and the appropriate type-matching variables between parenthesis. Function values can be stored in assignment of variables. For example, a simple add function:

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

int main() {
    int a;
    int b;

    a = add(39, 3);
```

```
    b = add(12, 5);

    print(a);             /* 42 */
    print(b);             /* 17 */

    return 0;
}
```

Note that the types of variables `a` and `b` must match with the function type of `add`. In this case, they are all integers.

3.5 Advanced Topics

The subjects covered below form a core part of what makes C? special as a language. If you are interested in how to do more powerful data manipulations in C?, we encourage you to read the following tutorial chapters!

3.5.1 Structs

Structs - short for structures - provide a means for users to create their own types out of clusters of primitive types. Structs require a struct definition that explains what is in the struct and how it is named. A struct can be created the same way as an array. Each parameter of a struct can then be assigned to unique values. For example:

```
struct foo {
    int i;
    float f;
}


int main()
{
    struct foo foo;
    struct foo foo2;
    struct foo[] foo_arr;

    foo_arr = make(struct foo, 1);
    foo = make(struct foo);

    foo.i = 1;
    foo.f = 3.14;
    print(foo.i);               /* 1 */
    print_float(foo.f);         /* 3.14 */

    foo_arr[0] = foo;
```

```
    foo2 = foo_arr[0];
    print(foo_arr[0].i);        /* 1 */
    print_float(foo2.f);        /* 3.14 */

    return 0;
}
```

Above, we create a new type `foo` and then instantiate two implementations of that type, `foo, foo2`. We also create an array of type `foo`, showing that it is possible to create arrays of structs (i.e. a struct type is treated like any other type). Because structs are simply user defined types, it is possible to assign one struct to another of the same struct type. Finally, struct access is accomplished using the `x.y` syntax, where x is the struct variable and y is the struct field we want to access.

Structs allow for the development of powerful libraries that rely on specific struct objects. Below, we show how structs can be used to create a simple point type for 2d distance calculations.

```
struct point {
    int x;
    int y;
}

int manhattan_distance(struct point a, struct point b) {
    int x_diff;
    int y_diff;

    if ( a.x > b.x ) {
        x_diff = a.x - b.x;
    } else {
        x_diff = b.x - a.x;
    }

    if ( a.y > b.y) {
        y_diff = a.y - b.y;
    } else {
        y_diff = b.y - a.y;
    }

    return x_diff + y_diff;
}
```

The `point` struct definition acts as a predefined object. This can then be combined with functions that can manipulate that object and its properties.

C? is also capable of assigning functions to structs, resulting in an inheritance-less form of object oriented programming. Struct functions, or methods, require a struct definition in brackets to use as the attached struct. Unlike a tradtional function definition, a method then requires the name of the function, followed by the return type and arguments. The bound struct can then be used as if it were a passed in parameter.

Below, we present the same 2d distance calculation code with methods. We also show how to call methods.

```
struct point {
    int x;
    int y;
}

/* Use s as a bound struct of type point */
[struct point s] manhattan_distance(struct point b) int {
    int x_diff;
    int y_diff;

    if ( s.x > b.x ) {          /* s can be used as if it were passed in */
        x_diff = s.x - b.x;
    } else {
        x_diff = b.x - s.x;
    }

    if ( s.y > b.y) {
        y_diff = s.y - b.y;
    } else {
        y_diff = b.y - s.y;
    }

    return x_diff + y_diff;
}

int main() {
    struct point a;
    struct point b;

    a = make(struct point);
    b = make(struct point);

    a.x = 5;
    a.y = 10;
    b.x = 10;
    b.y = 15;

```

```
    print( a.manhattan_distance(b) );  /* The manhattan_distance method uses */
                                       /* a in place of s */
                                       /* This code therefore prints 10 */
    return 0;
}
```

Note that users can define the same function name to different structs. For example, there can be a `manhattan_distance` method for type `point` and a different `manhattan_distance` method for, e.g., type `vector`.

Note that although it is possible to nest structs, it is not possible to call access structs N layers deep. Instead, one needs to create a variable reference to the inner struct, like so:

```
struct bar {
    int i;
}

struct foo {
    struct bar;
}

...

foo.bar.i;  // Is NOT allowed.
bar = foo.bar;
bar.i;      // IS allowed.
```

Like arrays, structs are heap objects. See the Memory Management section below for more.

3.5.2 Matrices

Although C? does not support nested arrays, users can still create matrix types in order to do complex data analysis. C? links directly to the C Eigen library and makes available a significant subset of Eigen matrix operations. Because Eigen is a low level C library, matrix operations in C? are fairly fast. Below is a simple example of initializing float matrices and using matrix operations:

```
int main(){
    fmatrix fm1;
    fmatrix fm2;
    fmatrix fm3;

    /* Create a 5 by 5 matrix of zeros */
```

```
    fm1 = init_fmat_zero(5, 5);
    /* Create a 5 by 5 matrix of 2.5's */
    fm2 = init_fmat_const(2.5, 5, 5);

    /* Matrix literal */
    fm3 = [[1.0, 2.0, 3.0], [4.0, 5.0, 60], [7.0, 8.0, 9.0]];

    print_mat((fm1 + 1.0) + fm2);
    fm1 = fm1 + 1.0;
    print_mat((fm1 + 12.0) .. fm2);    /* Matrix multiplication */
    print_mat(fm1 * fm2);                   /* Hadamard product */

    return 0;
}
```

The corresponding output of this C? code is:

```
3.5 3.5 3.5 3.5 3.5
3.5 3.5 3.5 3.5 3.5
3.5 3.5 3.5 3.5 3.5
3.5 3.5 3.5 3.5 3.5
3.5 3.5 3.5 3.5 3.5
162.5 162.5 162.5 162.5 162.5
162.5 162.5 162.5 162.5 162.5
162.5 162.5 162.5 162.5 162.5
162.5 162.5 162.5 162.5 162.5
162.5 162.5 162.5 162.5 162.5
2.5 2.5 2.5 2.5 2.5
2.5 2.5 2.5 2.5 2.5
2.5 2.5 2.5 2.5 2.5
2.5 2.5 2.5 2.5 2.5
2.5 2.5 2.5 2.5 2.5
```

Note that matrix operations are by default element-wise. For example, `(fm1 + 1.0)` increments every element in `fm1`. To see the full list of Matrix operations available, please review the Language Reference Manual.

Like arrays, Matrices are heap objects. See the Memory Management section below for more.

3.5.3 Memory Management

Structs, Arrays, and Matrices are all heap objects - they are stored on the heap, the run time manipulates pointers to these objects, and as a result they are pass by reference. Because these objects are all stored on the heap, references to them will eventually need to be freed. For short programs and scripts, we

recommend ignoring memory management concerns. For larger scripts, especially those using many matrix operations, we recommend manually freeing the memory.

This can be done with the free command for matrices and structs, and the free_arr command for arrays. Both take a reference to an object and frees the associated memory. An example (from the DEEP library) can be seen below:

```
float quadratic_cost(fmatrix x, fmatrix y) {
  float ret;
  fmatrix fm;
  fm = x - y;
  ret = square(l2_norm(fm)) * .5;
  free(fm);
  return ret;
}
```

Note that every matrix operation allocates new memory, as all matrix operations clone the initial matrix instead of doing operations in place. Thus, any time matrix variables go out of scope, they should be manually freed.

3.5.4 Function Pointers

In many cases, it can be useful to abstract how a function is called away from what the function does. C? supports using function pointers as a way to provide this abstraction. Function pointers allow references to functions to be passed as arguments, much like any other value. This in turn provides a means for users to create highly generalized yet powerful libraries with plug-and-play modular components that can be user specified.

A function pointer type is defined by the types of the arguments the function takes and the return type. An example is provided below:

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

int mult(int x, int y) {
    return x * y;
}

/* In the function pointer type below, the last value type is the return */
void print_bin(fp (int, int, int) f, int x, int y) {
    print(f(x, y));
    return;
}
```

```
14
15  int main() {
16      print_bin(add, 7, 35);        /* 42 */
17      print_bin(mult, 7, 6);        /* 42 */
18
19      return 0;
20  }
```

Note that function pointers can also be combined with structs to create abstract interfaces that can be easily extended by end users for a variety of domain specific applications.

3.5.5 Links to C

Although C? is a powerful language, there are many libraries and features available in C that are not available in C? (e.g. pointer manipulation, direct memory management, etc). In order to make C? as flexible as possible for a wide variety of use cases, C? supports direct linking with C. In effect, users can write functions in C and use them directly in C?. This is done with the `extern` keyword, as shown below:

```
1   extern void printbig(int c);
2
3   int main() {
4       printbig(72); /* H */
5       return 0;
6   }
```

`printbig` is a c function defined as below. Note that parts of the `printbig` code are left out.

```
1   void printbig(int c) {
2       int index = 0;
3       int col, data;
4       if (c >= '0' && c <= '9') index = 8 + (c - '0') * 8;
5       else if (c >= 'A' && c <= 'Z') index = 88 + (c - 'A') * 8;
6       do {
7           data = font[index++];
8           for (col = 0 ; col < 8 ; data <<= 1, col++) {
9               char d = data & 0x80 ? 'X' : ' ';
10              putchar(d); putchar(d);
11          }
12          putchar('\n');
13      } while (index & 0x7);
14  }
```

In order to have the compiler 'see' `extern`ed C code, the appropriate `.c` file needs to be placed in `/lib/src/`.

3.6 Conclusion

We hope you enjoyed this short tutorial on how to use the C? language! While we described many powerful features of C?, we only scratched the surface of how these features can be combined and applied. Take a look at the full Language Reference Manual for a formal review of all C? features. Happy coding!

4 C? Language Reference Manual

4.1 Introduction

This Language Reference Manual describes C?, a multi-paradigm imperative general purpose language, as well as C? DEEP, a deep learning library built on top of the C? language. Following the theory that powerful domain application comes from a strong general foundation, C? is to be flexible and easily extended. Features defining the language include strong typing, built in matrix operation support, no-inheritance object oriented structs, and function pointers. The following sections delineate in detail the types, conventions, syntax, program structure, operations, and libraries included in the C? language.

4.2 Data Representation

Types define the various formats of data. Primitive types represent fundamental building blocks that have absolute values associated with them; nonprimitive types represent types that are compositions of primitive types or references to primitive types. Variables and functions must have a type associated with them for semantic correctness. All relevant operations must be type-checked for semantic correctness. Types may have an associated literal value that can be represented in C?.

4.2.1 Types

Primitive Types

Type	Description	Example
`int`	`Integer`	`int i;`
`float`	`Float`	`float f;`
`bool`	`Boolean`	`bool b;`
`string`	`String`	`string s;`
`void`	`Empty Type`	`void foo() {}`
`fmatrix`	`nxm Float Matrix`	`fmatrix fm;`

Nonprimitive Types

Type	Description	Example
`struct ID { type ID; ...}`	`Struct`	`struct s {int i; int f;}`
`typ[]`	`Array of typ`	`float[] f_arr;`
`fp (typ, typ...return typ)`	`Function Pointer`	`fp (int, int, void) p;`

4.2.2 Literals

Examples of each literal type are presented below:

Types and Corresponding Literal Examples

Type	Literal
int	42
float	42.0
bool	true
string	"Hello World!"
fmatrix	[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]
array	(int[]) {1, 2, 3, 4}
fp	int foo(int a, int b) { return 1; }

The regexes used for each literal are below:

Types and Corresponding Literal Regexes

Type	Regex
int	['0'-'9']+
float	(['0'-'9']+'.'['0'-'9']* | ['0'-'9']*'.'['0'-'9']+)
bool	true|false
string	"([' '-'!' ''-'[' ']'-' '])*"

Array literals, matrix literals, and function literals rely on parsing semantics. An array literal is defined as follows:

`(type[]) {expr, expr, expr...}`

where an `expr` is a series of expressions defined in the Expressions section below, and `type` is one of the types defined in the Type section above.

Matrix literals follow intuitive nested array definitions. A matrix literal is defined as:

`[ARRAY, ARRAY...]`
where each ARRAY is a float or int array of the same length and type, defined as comma separated ints or floats between square brackets.

Function literals are simply function definitions. Function definitions are explained in depth below. They are defined as:

`type ID(type ID, type ID...) { var_decls stmt_list }`

where `var_decls` is a series of variable declarations, `stmt_list` is a list of statements that resolve to control flow and expressions, and `type` is one of the types defined in the Type section above.

4.3 Lexical Conventions

4.3.1 Spacing

The following characters will be treated as whitespace: space, tab, line return. These will be ignored, but they will separate adjacent identifiers, literals, and keywords that might otherwise be used as a single identifier, literal, or keyword.

4.3.2 Comments

Comments are used to explain C? code and will be ignored by the C? compiler. Comments can be delineated as follows: `/* this is a comment */`.
Comments can span multiple lines using the same syntax.

4.3.3 Identifiers

An identifier in C? is a programmer defined object. The identifier starts with a character, and is composed of alphanumeric characters and underscore. Specifically, an identifier matches the regex:

```
['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
```

4.3.4 Keywords

C? has a set of keywords that carry special meaning and cannot be used as identifiers. In addition to the table below, any of the types described in the Type section above are also keywords and cannot be used as an identifier.

Keywords and Purpose

Keyword(s)	Purpose
if, else	Control flow for if/else branching
while, for	Control flow for looping
return	Control flow for leaving a function
extern	Indicates a function definition that is defined in C

There are numerous built-in functions described in the Built-In Functions section below. Though these functions are not keywords, users cannot define other functions with the same name. For example, the `print` function is built into C?; users can define variables named print, but cannot create a new global function named print. Users *can* however define functions with the same name as built in functions provided they are attached to a struct (see the Structs section below).

4.4 Program Structure

4.4.1 Scoping Rules

C? uses static scoping. The scope of an object is limited to the block in which it is declared, and overrides the scope of an object with the same identifier

declared in a surrounding block. In other words, an identifier will map to the closest definition. Scopes are created by enclosed curly braces. Thus, if/else blocks, for/while loops, and struct/function definitions all create new scopes.

Note that all local variable declarations in C? must occur at the start of a function definition (see the Declarations section below). Thus, although if/else and for/while blocks create new scopes, new variables cannot be defined in those scopes. Further, C? does not allow nested function definitions. Scoping rules for identifier availability apply primarily when dealing with globally scoped identifiers such as variables or function names, or when dealing with nested structs. An example of various scoping rules is below:

```
int i;              /* i1 */

struct foo {
    int i;
}

struct bar {
    int i;
    struct foo inner_foo;
}

void f() {
    i = 5;          /* Sets i1 */
    return i;
}

int main() {
    int i;          /* i2 */
    struct foo foo; /* i3 */
    struct bar bar; /* i4 */

    foo = make(struct foo);
    bar = make(struct bar);

    i = 42;          /* Sets i2 */
    foo.i = 2;       /* Sets i3 */
    bar.foo = foo;
    bar.i = 3;       /* Sets i4 */

    print(f());      /* 5 */
    print(i);        /* 42 */
    print(foo.i);    /* 2 */
    print(bar.i);    /* 3 */

    return 0;
```

```
}

Note in the above example that each definition of the integer \texttt{i} refers
 ↪  to a different value.
```

4.4.2 Declarations

Declarations tell the program which local identifiers to track for a given scope. Regardless of the type of the identifier, the declaration for the identifier must be at the top of the scope it is in. For all declarations besides function declarations, value assignment is separated.

Struct Type and Struct Declarations A struct type declaration requires the following pattern:

`struct NAME { type ID1; type ID2; ...}`

where `NAME` is the identifier of the struct type and each `ID` is the identifier of a different struct member of type `type`. For example, a point type could look like:

```
    struct point {
        int x;
        int y;
        string name;
    }
```

Note that struct types can include nested struct members as well as arrays and function pointers. Further, note that struct type declarations must occur in global scope before the type is used in any function definition.
To declare a struct variable of a struct type, the following pattern is used:

`struct NAME ID;`

where `NAME` is the identifier of the struct type and `ID` is the identifier of the variable. Because structs are nonprimitives, a reference must be created to the struct memory location before the struct can be used. This is done with the `make` keyword, with the following pattern:

`ID = make(struct NAME);`

where `ID` is the identifier for the struct variable and `NAME` is the identifier for the struct type. An example point variable declaration and initialization could look like:

```
struct point p;
p = make(struct point);
```

Function Declarations A function declaration requires the following pattern:

`type NAME ( type ID, type ID, ...) { STMTS }`

where `NAME` is the identifier of the function, `ID` is the identifier for an argument of the function with type `type`, and `STMTS` are a series of zero or more program statements, including declarations, control flow operations, and expressions.

Functions must be declared in global scope. Unlike other declarations, the definition of a function is provided during declaration. Further, functions cannot be overloaded or replaced once named. An example function declaration may look like:

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

The `int main() {}` function declaration is a unique function that must be included in every runnable C? file, as it defines the execution entry point.

Functions written in C can be used in a C? program through the use of the `extern` keyword. An `extern`ed function definition follows the same rules as a normal function definition, but there are no `STMTS` and the `extern` keyword is included at the front of the definition. External functions use the following syntax:

`extern type NAME ( type ID, type ID, ...);`

Variable Declarations A primitive variable declaration requires the following pattern:

`type ID;`

where `ID` is an identifier for a variable of type `type`. An example variable declaration may look like:

```
int x;
float y;
```

```
    string s;
    bool b;
```

Array declarations require the following pattern:

`type[] ID;`

where `ID` is an identifier for a variable of type `type` and `[]` indicates the variable is an array, i.e. a collection of individual components of type `type`. Array types can include structs, function pointers, and primitives. An example may be:

```
    int[] i_arr;
    float[] f_arr;
    struct foo[] str_arr; /* for some struct foo */
```

Function pointer declarations require the following pattern:

`fp (type, type, ... r_type) ID`

where `type` refers to a type of a function argument, `r_type` refers to the return type of a function, and `ID` is the identifier for the function pointer variable. An example may be:

```
    fp (int, int, void) p; /* for some function that takes two ints
                              and returns void */
```

Multiple variables of the same type can be declared simultaneously, using the following pattern:

`type: ID, ID, ID...;`

where `ID` is an identifier for a variable of type `type`.

4.4.3 Control Flow

Control flow in C? defines where the current execution point is and how it transfers between different blocks of code. The default control flow is top-to-bottom sequential - lines of code will execute sequentially from top to bottom unless one of the following constructs is used. Control flow will always start at the top of the `int main` function block, and will end once the `int main` function block finishes.

If, Else. If/Else control allows a user to specify which of two branches of code to follow given the value of a boolean condition. If the condition resolves to a true value, control will pass to code in the If block. If the condition resolves to a false value, control will pass to code in the Else block. Code execution in If/Else constructs are mutually exclusive; if the If block is being run, execution will skip over the Else block and vice versa. Within either the If or the Else block, code executes as per default: top-to-bottom sequential.

An If/Else construct follows the following patterns:

```
if ( EXPR ) { STMTS } else { STMTS }
if ( EXPR ) { STMTS }
```

where `EXPR` is a boolean expression that resolves to true or false and `STMTS` is a list of zero or more commands that can include assignment and other control flow. Note that it is therefore possible to nest If/Else constructs. Note too that the Else block is not strictly required.

An example If/Else construct may look like:

```
1  int x;
2  x = 5;
3  if (x < 10) {
4      print_string("x is less than 10");
5      print_string("leaving if block");
6  } else {
7      print_string("x is greater than or equal to 10");
8  }
9  print_string("outside of if/else");
```

In the above example, the boolean condition will resolve to true as the variable `x` is less than 10. Thus, control will pass to the If block. Execution will then proceed sequentially, first printing `x is less than 10` followed by `leaving if block`. Control will then skip over the else block entirely, continuing in sequential order again from the last print statement. It will finally print `outside of if/else` before completing.

While, For. While loops and for loops allow a user to specify repetitions of a certain block of code given the value of a boolean condition. As long as the boolean condition resolves to true, the code inside the while/for loops will execute in top-to-bottom sequential order. After reaching the end of a loop block, the boolean condition is checked again. If the condition remains true, control jumps to the top of the loop block and repeats. If the condition is false, control leaves the loop block and continues in top-to-bottom sequential order.

A while loop follows the following pattern:

`while ( EXPR ) { STMTS }`
A for loop follows the following pattern:

`for ( OPTEXPR ; EXPR ; OPTEXPR ) { STMTS }`
In both cases, `EXPR` is a boolean expression that resolves to true or false, and `STMTS` is a list of zero or more commands that can include assignment and other control flow. Note that it is therefore possible to nest loops. In the for loop case, `OPTEXPR` is an optional expression that can include assignment or other operations. The first OPTEXPR is resolved before the loop starts; the second is resolved at each loop state.

Examples of both loops are shown below:

```
1  int i;
2  i = 0;
3
4  while (i < 10) {
5      i = i + 1;
6      print(i);
7  }
8
9  for (i = 0; i < 10; i = i + 1) {
10     print (i);
11 }
```

Control starts at the top of the program by initializing integer `i` to 0. Once control reaches the while block, the boolean condition `i < 10` is checked. Since 0 is less than 10, control enters the while block and proceeds in top-to-bottom sequential order. First the integer `i` is incremented by 1; then `i` is printed out. At the end of the while block, control returns to the top of the block and checks the boolean condition again. Since `i = 1 < 10`, the loop continues again. This repeats 10 times.

Once `i >= 10`, control leaves the while block and continues sequentially to the for block. Control executes the first OPTEXPR, which resets integer `i` to 0. Control then checks the boolean expression `i < 10`, which resolves to true. Since the boolean expression resolves to true, control passes to the inside of the for block and proceeds sequentially - in this case, printing out the value of `i`. Once control reaches the end of the for block, it executes the second OPTEXPR and increments `i` by 1. It then checks the boolean expression again to determine if the control should exit the for block or loop again. Since `i = 1 < 10`, the loop continues again. This repeats 10 times.

Note that the two loops above are equivalent. Further, note that all for loops can be rewritten as while loops and vice versa.

Call, Return. Function calls allow control to jump to a previously defined function declaration; return allows control to jump back to where the function call was made.

Function calls follow the following pattern:

`NAME(EXPR, EXPR...);`
where `NAME` is the identifier for a previously defined function and `EXPR` is an expression that evaluates to a value of the type required by the function definition of function `NAME`.

Return follows the following pattern:

`return OPTEXPR;`
where `OPTEXPR` is an optional expression that evaluates to a value with the same type as the return type of the encapsulating function. The value that is returned takes the place of the location of the function call. Because of returning values, a function call is also a type of expression, described in the Expressions section below.

For example:

```
1  int add(int x, int y) {
2      return x + y;
3  }
4
5  int main() {
6      print(add(5, 10));      /* 15 */
7      return 0;
8  }
```

4.5 Expressions

Expressions represent the lowest possible level of commands that the C? language is based on. Expressions either resolve to a value of a given type or act as assignment. Note that a function call is also an expression in that it resolves to a value.

4.5.1 Primary Expressions

The following expressions are primary expressions (building blocks for more complex components):

- All literals listed in the Literals section above.
- All identifiers described in the Identifier section above.
- Parenthetical expressions, i.e. `(expression)`.

4.5.2 Assignment

Assignment in C? takes two forms: equals (=) assignment, and pipe (=¿) assignment.

The former is done as a single command, where a single variable identifier is to the left of an 'equals' (=) operator with an expression of some sort on the right. The expression must resolve to a value of the same type as the variable on the left hand side. In other words, the equals operation is a left operand assignment where the identifier on the left of the operator is set to the expression on the right. For example:

```
int: x, y, z;
float: a, b, c;

/* Literal assignment */
y = 5;
z = 10;
b = 5.0;
c = 10.0;

/* Expression assignment */
x = y + z;  /* 15 */
x = y + c;  /* type error */
```

Pipe assignment is a unique component of C? that makes chaining commands easier and syntactically cleaner. Pipe assignment is a right operand assignment where the value/variable on the left is 'piped' into a function call as the first argument in the function. Pipes can be combined for multiple levels of function calls. An example is shown below:

```
int x;

int add(int a, int b) {
    return a + b;
}

x = 5 => add(5) => add(10);
x = add(add(5, 5), 10); /* Equivalent */

print(x);   /* 20 */
```

Note that the pipe operator can also be used to end lines, resulting in stylistically cleaner code. For example:

```
x = 5 =>
add(5) =>
add(10);
```

4.5.3 Arrays

Arrays are collections of data of a single type, stored in sequential memory. Arrays are heap objects that are pointers to a block of memory, and therefore are pass-by-reference. Formatting for declaration of variables and literals can be found in the Types and Literals sections above respectively.

In order to use an array, memory must be allocated for it on the heap using the make command as follows:

```
maketype, len
```

where `type` is the array type (see the Types section above) and `len` is the initial size of the array. Note that though the array is allocated, there are no values stored in place. Array memory can be freed using the free_arr function.

Arrays can be indexed and individually assigned with the following syntax:

```
arr[INDEX] = EXPR;
```

where `arr` is an identifier for an array variable, `INDEX` is an integer value less than the length of the array, and `EXPR` is an expression that resolves to a value of the type of the `arr` variable. Indexed array values can be used as part of C? expressions.

There are numerous built-in functions that work with arrays, such as `len`, `concat` and `append`. For more on how to use these functions, please see the Built-In Functions section below.

4.5.4 Structs

Structs allow users to create custom types out of clusters of primitive types. Structs are heap objects that are pointers to a block of memory, and are therefore pass-by-reference. Formatting for declaration of variables and literals can be found in the Types and Literals sections above respectively.

In order to use a struct, memory must be allocated for it on the heap using the make command as follows:

```
makestruct type
```

where `type` is the name of a struct type. Note that though the struct is allocated, there are no values stored in place. Struct memory can be freed using the free function.

Struct members can be accessed and individually assigned with the following syntax:

```
foo.member = EXPR;
```

where `foo` is a struct identifier, `member` is a struct member name for the struct type of identifier `foo`, and EXPR is an expression that resolves to a value of the type of member `member`. Struct member values can be used as part of C? expressions. Note that struct members may be other structs. However, it is not possible to nest struct access calls; instead, a new struct variable must be assigned and accessed. For example:

```
1  struct bar {
2      int i;
3  }
4
5  struct foo {
6      struct bar;
7  }
8
9  ...
10
11 foo.bar.i;  // Is NOT allowed.
12 bar = foo.bar;
13 bar.i;      // IS allowed.
```

C? handles method dispatch for structs, allowing methods to be assigned to a struct namespace. Struct functions, or methods, can be defined with the following pattern:

```
[struct s_type VAR] F_ID(type ID, ...)  r_type { STMTS }
```

where `s_type` is the name of the struct type, `VAR` is the name of the attached struct in the method, `F_ID` is the name of the method in the struct namespace, `type` is an argument type, `ID` is an identifier for a passed in argument, `r_type` is the return type of the method, and `STMTS` are C? statements that include

control flow. Methods can be called with the following syntax:

`foo.method(ID, ID...)`

where `foo` is a struct variable, `method` is an attached method name, and `ID` is a name of an argument passed into the method. An example of defining and calling a method is shown below:

```
struct point {
    int x;
    int y;
}

/* Use s as a bound struct of type point */
[struct point s] manhattan_distance(struct point b) int {
    int x_diff;
    int y_diff;

    if ( s.x > b.x ) {          /* s can be used as if it were passed in */
        x_diff = s.x - b.x;
    } else {
        x_diff = b.x - s.x;
    }

    if ( s.y > b.y) {
        y_diff = s.y - b.y;
    } else {
        y_diff = b.y - s.y;
    }

    return x_diff + y_diff;
}

int main() {
    struct point a;
    struct point b;

    a = make(struct point);
    b = make(struct point);

    a.x = 5;
    a.y = 10;
    b.x = 10;
    b.y = 15;

    print( a.manhattan_distance(b) );  /* The manhattan_distance method uses */
                                       /* a in place of s */
```

```
                                    /* This code therefore prints 10 */
    return 0;
}
```

Note that users can define the same function name in different structs. For example, there can be a `manhattan_distance` method for type `point` and a different `manhattan_distance` method for, e.g., type `vector`.

4.5.5 Matrices

C? supports matrices as well as a wide array of built-in matrix operations. Matrices are heap objects that are pointers to a block of memory, and therefore are pass-by-reference. Formatting for declaration of variables and literals can be found in the Types and Literals sections above respectively.

In order to use a matrix, memory must be allocated for it on the heap using one of the following commands:

```
init_fmat_const(float VAL, int ROW, int COL)
init_fmat_zero(int ROW, int COL)
init_fmat_identity(int ROW, int COL)
```

where `VAL` is an initial value (set to 0 for `init_fmat_zero`), `ROW` is an integer number of rows in the matrix, and `COL` is an integer number of columns in the matrix. Matrices can also be initialized directly with a matrix literal:

```
 [[1.0, 2.0, 3.0], [1.1, 1.2, 1.3] ]
```

Once created, the matrix will have initial values stored. Matrix memory can be freed using the `free` function. To see other methods related to matrices, please see the Built-In Functions section below.

Matrices can be indexed and assigned as follows:

```
fmat[R, C] = EXPR;
```

where `fmat` is the identifier of a matrix, `R` and `C` are integers denoting the row and column of the value being indexed, and `EXPR` is an expression that resolves to a value of the same type as the matrix. Indexed matrix values can be included in C? expressions.
Like arrays, matrices have many built in functions that can be found in the Built-In Functions section below. Matrices also have specific matrix operators that can be found in the Operators section below. Note that each matrix operation creates a new matrix clone with its own allocated memory that needs to

eventually be freed. Indexing, assigning, and passing matrices does *not* create a new matrix, but will modify the old matrix in place (due to pass-by-reference). The `copy()` method can be used to duplicate an existing matrix.

4.5.6 Function Pointers

Function pointers in C? allow references to functions to be passed as arguments and called. Formatting for declaration of variables and literals can be found in the Types and Literals sections above respectively. Function pointers are (obviously) pass-by-reference with regards to the target function.

4.5.7 Operators

Below are the tables describing the various built in operators in the C? language. If a table specifies certain types, all operations in that table are assumed function in the way specified for only those types.

Unary

Operator	Explanation
`( -expr )`	`Defined for int/float expressions.  Numeric negation.`
`( !expr )`	`Defined for boolean expressions.  Logical negation.`

Arithmetic (int, float)

Operator	Explanation
`( expr1 + expr2 )`	`Numeric sum.`
`( expr1 - expr2 )`	`Numeric subtraction.`
`( expr1 * expr2 )`	`Numeric multiplication.`
`( expr1 / expr2 )`	`Numeric division.`

Matrices (fmatrix)

Operator	Explanation
`( expr1 + expr2 )`	`Element-wise matrix sum.`
`( expr1 - expr2 )`	`Element-wise matrix subtraction.`
`( expr1 * expr2 )`	`Element-wise matrix multiplication.`
`( expr1 / expr2 )`	`Element-wise matrix division.`
`( expr1 ..  expr2 )`	`Matrix multiplication.`
`( expr1 )`	`Matrix Transpose.`

Note: If one and only one of either `expr1` or `expr2` are scalars instead of matrices for the first four binary operators (`+, -, *, /`, we return an element-wise operation with that scalar.

Assignment

Operator	Explanation
(expr % expr)	Equals assignment. See Assignment for more.
(expr => expr)	Pipe assignment. See Assignment for more.

Relational

Operator	Explanation
(expr1 < expr2)	Less than comparison.
(expr1 > expr2)	Greater than comparison.
(expr1 <= expr2)	Less than or equal to.
(expr1 >= expr2)	Greater than or equal to.

Equality

Operator	Explanation
(expr1 == expr2)	Equality comparison.
(expr1 != expr2)	Inequality comparison.

Logical

Operator	Explanation
(expr1 && expr2)	Logical AND.
(expr1 || expr2)	Logical OR.

4.5.8 Operator Precedence

Operators are ordered with the following precedence rules from top to bottom, highest to lowest precedence.

Precedence

Operator	Name
(expr)	Parenthetical statements.
(expr)^	Matrix transpose.
({- !} expr)	Negation operations.
(expr {* / ..} expr)	(Matrix) Multiplication, Division operations.
(expr {+ -} expr)	Addition, Subtraction operations.
(expr {>, <, >=, <=} expr)	Comparison operations.
(expr {== !=} expr)	Equality operations.
(expr && expr)	Logical AND.
(expr || expr)	Logical OR.
(expr => expr)	Pipe assignment.
(expr = expr)	Equal assignment.

4.6 Built-in Functions

The following functions are general functions are built into the C? language.

- `void printf(str, type, type type...);` An external call to the C

printf function. Takes a format string and a variable number of arguments based on the format string.

- `int time();` Returns the current second time.

The following functions are type conversion functions built into the C? language.

- `float float_of_int(int i);` Type conversion from int to float.
- `int int_of_float(float f);` Type conversion from float to int.

The following functions are memory management functions built into the C? language.

- `struct foo make(struct foo);` Allocates memory for a struct of type `foo`.
- `type[] make(type[], int i);` Allocates memory for a struct of type `type` of size `i`.
- `void free(fmatrix fm);` Frees allocated memory for a matrix.
- `void free(struct foo);` Frees allocated memory for a struct.
- `void free_arr(type[] arr);` Frees allocated memory for an array.

The following functions are array functions built into the C? language.

- `int len(type[] arr);` Length of passed in array.
- `type[] append(type[] arr, type id);` Appends `id` to array `arr`.
- `type[] concat(type[] arr1, type[] arr2);` Concatenates `arr2` to the end of `arr1`.

The following functions are matrix functions built into the C? language.

- `int cols(fmatrix fm);` Returns the number of columns in a matrix.
- `int rows(fmatrix fm);` Returns the number of rows in a matrix.

4.7 Libraries

Below we list the packaged libraries that come with the C? language. For each library, we highlight a few important functions and structs, but we do not necessarily cover every function or struct that is available in the library in this document. For further detail on the available functions in a given library, please examine the library source available in the Appendix.

4.7.1 Interaction with Compiler

Libraries are implemented by appending the contents of a given library to the top of a compiled C? program during compilation. In order to allow the compiler to see a written library, the library source must be located in the `/lib/` folder.

4.7.2 IO

The following functions are included in the standard IO library.

- `void flush();` Flushes standard out.
- `void print(int i);` A helper function that specifies printf to ints only. Will take in and print an integer type.
- `void printb(bool b);` A helper function that specifies printf to booleans only. Will take in and print a boolean type.
- `void print_float(float f);` A helper function that specifies printf to floats only. Will take in and print a float type.
- `void print_string(string s);` A helper function that specifies printf to strings only. Will take in and print a string type.
- `void print_line();` A helper function that prints a new line.
- `void print_fmat_arr(fmatrix[] f_arr);` Prints an array of matrices.
- `void print_fmat_arr_dims(fmatrix[] f_arr);` Prints the dimensions of each matrix in a matrix array.

4.7.3 Math

The following functions are included in the standard Math library.

- `float sin(float x);` Calculates the `sin` value of the input.
- `float cos(float x);` Calculates the `cos` value of the input.
- `float log(float x);` Calculates the `log` value of the input.
- `float pow(float x, float y);` Calculates the value of `x` raised to the power `y`.
- `int modulo(int x, int y);` Calculates the value of `x` mod `y`.
- `int rand();` Returns a random number from 0 to INTMAX.
- `float rand_norm(float mu, float sigma);` Returns a random float according to the normal distribution defined by `mu` and `sigma`.

4.7.4 Eigen

The Eigen library is designed to augment the matrix operations available in C?. The following functions are included in the Eigen library.

- `void print_mat(fmatrix fm);` Prints a matrix.
- `fmatrix init_fmat_zero(int r, int c);` Creates a new matrix of dimensions `(r, c)` and fills it with zeros.
- `fmatrix init_fmat_const(float s, int r, int c);` Creates a new matrix of dimensions `(r, c)` and fills it with const `s`.
- `init_fmat_identity(int r, int c);` Creates a new matrix of dimensions `(r, c)` and makes it an identity matrix.
- `fmatrix map(fmatrix fm, fp (float, float) ptr);` Creates a new matrix where function `ptr` is applied to each element in matrix `fm`.

4.7.5 DEEP

The DEEP library is a deep learning library designed to make implementing deep learning models easier in C?. Currently, the library automates loading of the MNIST handwritten-digits binary, and supports making fully-connected feedforward architectures with arbitrary numbers of neurons and layers. Due to the complexity of this library, individual components are discussed in slightly more detail, especially with regard to common usage.

```
fmatrix[]: train_fm_images, train_fm_labels, test_fm_images, test_fm_labels;

train_fm_images = make(fmatrix, 50000);
train_fm_labels = make(fmatrix, 50000);

test_fm_images = make(fmatrix, 10000);
test_fm_labels = make(fmatrix, 10000);

/* Load train */
load_mnist_data(train_fm_images, train_fm_labels,
  "mnist_data/train-images-idx3-ubyte",
  "mnist_data/train-labels-idx1-ubyte"
);

/* Load test */
load_mnist_data(test_fm_images, test_fm_labels,
  "mnist_data/t10k-images-idx3-ubyte",
```

```
    "mnist_data/t10k-labels-idx1-ubyte"
  );
```

load_mnist_data

MNIST is a classic machine learning problem that involves learning to recognize a set of handwritten digits. The MNIST training corpus consists of 60,000 pieces of labeled training data, and 10,000 of pieces labeled test data. All images are 28x28 grayscale images, with each pixel value encoded a]s an integer within the range [0, 255].

The `load_mnist_data` library function automatically loads the MNIST binary from `http://yann.lecun.com/exdb/mnist/` into the fmatrix arrays which are provided as arguments. For ease of use, it reformats the 28x28 images into 784x1 fmatrices, and it uses one-hot-encoding to convert the number label (an integer from {0,1,...,9}) into a sparse 10x1 matrix, where all values are 0 except for the corresponding label index, which is 1. For example, 1 would be encoded as [0, 1, 0, 0, 0, ...].

This library function allows users to quickly plug into a well-known dataset so they may test the other features of our learning library on actual real-world data.

```
struct fc_model {
  fmatrix[] train_x;
  fmatrix[] train_y;
  fmatrix[] test_x;
  fmatrix[] test_y;
  fmatrix[] biases;
  fmatrix[] weights;
  int[] layer_sizes;
  int epochs;
  int mini_batch_size;
  float learning_rate;
  fp (float) weight_init;
  fp (float, float) activate;
  fp (float, float) activate_prime;
  fp (fmatrix, fmatrix, float) cost;
  fp (fmatrix, fmatrix, fmatrix, fmatrix) cost_prime;
}
```

struct fc_model

The `fc_model` struct is the primary component of deep.mc. Here we define an API which allows users to easily implement a fully-connected feed-forward

model. All they need to do is populate the fc_model struct fields, which are defined as follows:

- `train_x`: an array of fmatrices which represent the training corpus
- `train_y`: an array of fmatrices that are the labels corresponding to train_x
- `test_x`: an array of fmatrices which represent the test corpus
- `test_y`: an array of fmatrices that are the labels corresponding to test_x
- `biases` and weights: an array of fmatrices representing the bias and weight parameters of the network. The user does not need to ever touch these; the deep library has its own logic for initializing and optimizing these parameters.
- `layer_sizes`: and integer array describing the network architecture. The length of the array represents the number of layers in the model, and the value at index i denotes the number of neurons in layer i. Note: this array must be of length ¿= 2, because every neural needs at least an input and output layer. Additionally, the user will need to size the input layer correctly to match the dimensions of their training data.
- `epochs`: How many epochs to train for
- `mini_batch_size`: The mini_batch_size. Must be ¡= the size of the training corpus.
- `learning_rate`: the alpha coefficient used during backprop to update weight and bias values.
- `weight_init`: a function pointer used to initialize the bias and weight parameters.
- `activate`: a function pointer determining the activation function to be used at each neuron.
- `activate_prime`: a function pointer to the derivative of the activation function.
- `cost`: a function pointer to the cost function
- `cost_prime`: a function pointer to the derivative of the cost function

To quickly try out the fc_model struct on real-world data, the user need only load the mnist binaries using our `load_mnist_data` helper function, populate the struct fc model, and call `train()`.

cost and activation functions deep.cqm comes with the following cost functions:

- `float quadratic_cost(fmatrix x, fmatrix y);`
- `fmatrix quadratic_cost_prime(fmatrix z, fmatrix x, fmatrix y)`
- `float cross_entropy_cost(fmatrix x, fmatrix y)`
- `fmatrix cross_entropy_cost_prime(fmatrix z, fmatrix x, fmatrix y)`

and the following activation functions:

- `float sigmoid(float z)`
- `float sigmoid_prime(float z)`
- `float tanh(float z)`
- `float tanh_prime(float z)`
- `float relu(float z)`
- `float relu_prime(float z)`

train This is a function bound to the fc_model struct. After setting all the fields, the user may call `fc.train()` to train the model using vanilla back propagation for an arbitrary number of epochs.

backprop This function is the core of our fc_model interface; it defines the procedure for how we compute error terms throughout the network and update parameter values. The user should never need to call this themselves. Currently we only support the most basic backpropagation and gradient descent. Future work includes implementing more advanced trainers, such as adagrad or adam, which the user can then choose from. Additionally, we would have liked to open a function pointer interface from which the user could have supplied their own, custom trainer.

predict `fc.predict(X)` runs the current network on a single piece of data, and returns the output fmatrix.

evaluate `fc.evaluate()` runs the network on all of the test data supplied in `fc.test_x` and `fc.test_y`. It prints the overall performance and cumulative cost.
demo
`fc.demo(int n)` shows n examples of correct predictions and n examples of incorrect ones. The first incorrect prediction it shows is the one with lowest cost, i.e. the incorrect prediction the model was most confident about being correct.

misc In addition to the above, the following functions are included in the DEEP library.

- `int argmax(fmatrix fm);` Returns index of largest value in a 1 dimensional matrix, or row with the largest head in a 2 dimensional matrix.
- `float l2_norm(fmatrix fm);` Calculates the l2_norm of the passed in matrix.
- `float mat_sum(fm)` return the sum of all the elements within a column matrix
- `void print_mnist_image(fmatrix fm)` pretty-prints a 784x1 fmatrix representing an mnist image

Summary Putting it all together, using the fc_model struct looks like this:

```
int main()
{
  struct fc_model fc;
  fmatrix[]: train_fm_images, train_fm_labels, test_fm_images, test_fm_labels;
  int[] layer_sizes;
  int: epochs, mini_batch_size;
  float learning_rate;

  /* seed random number generator */
  srand(time());

  /* define hyperparameters */
  epochs = 20;
  learning_rate = .1;
  mini_batch_size = 10;
  layer_sizes = (int[]) {784, 50, 10};

  /* allocate memory */
  fc = make(struct fc_model);

  train_fm_images = make(fmatrix, 60000);
  train_fm_labels = make(fmatrix, 60000);

  test_fm_images = make(fmatrix, 10000);
  test_fm_labels = make(fmatrix, 10000);

  /* Load train */
  load_mnist_data(train_fm_images, train_fm_labels,
    "mnist_data/train-images-idx3-ubyte",
    "mnist_data/train-labels-idx1-ubyte"
  );
```

```
32
33    /* Load test */
34    load_mnist_data(test_fm_images, test_fm_labels,
35      "mnist_data/t10k-images-idx3-ubyte",
36      "mnist_data/t10k-labels-idx1-ubyte"
37    );
38
39    /* Popuate fc model fields */
40    fc.train_x = train_fm_images;
41    fc.train_y = train_fm_labels;
42    fc.test_x = test_fm_images;
43    fc.test_y = test_fm_labels;
44    fc.layer_sizes = layer_sizes;
45    fc.epochs = epochs;
46    fc.mini_batch_size = mini_batch_size;
47    fc.learning_rate = learning_rate;
48    fc.weight_init = norm_init;
49    fc.activate = sigmoid;
50    fc.activate_prime = sigmoid_prime;
51    fc.cost = cross_entropy_cost;
52    fc.cost_prime = cross_entropy_cost_prime;
53
54    fc.train();
55    fc.demo(5);
56
57    return 0;
58  }
```

5 Project Plan

5.1 Processes

5.1.1 Planning

Planning for C? took place in two major settings: weekly meetings with T.A. Kai-Zhan Lee, and bimonthly team meetings.

During weekly meetings the team set out goals for the upcoming week based on feedback from Kai-Zhan (both in terms of feasibility and implementation) and presented our progress on the prior week to the team and to the T.A. In this way we were able to constantly set progressing goals while simultaneously confirming our prior work.

Bimonthly team meetings were used as a way to realign the team on the final vision. These longer meetings included discussions about the inner workings of our language and the important features that needed to be developed for our final vision, as well as who would be in charge of developing a given feature. Bimonthly team meetings were also used as a way to catch other team members up to speed with the latest changes to the repository for any given feature, including how to use the feature and how to merge the feature into the main branch of the repository.

General day-to-day communication (mostly scheduling and short updates) occurred over Instant Messaging.

5.1.2 Specification

During early planning stages we developed a language specification that laid out the features and syntax of C?. Our original goal was to develop a language that could be used to demo the MNIST Hand Writing Identification deep learning task. To that end, we planned for C? to include structs, function pointers, and matrices. As C? evolved into a more general language, specification was updated to include external functions and arrays. Formal syntactic and lexical specification was laid out in the LRM. Feature specification was developed in tandem with the language tutorial.

5.1.3 Development

Development of C? was feature-centered. Features were assigned to a specific team member, and that team member was responsible for front-to-back development of the feature. This included scanning, semantic checking, and testing. Once features were completed, team members worked together to merge changes and create integration tests. Different features required different levels of work, and as such were assigned on a complete-as-you-go basis.

5.1.4 Testing

Before any feature was merged back into the main branch of the repository, it was tested with both positive and negative test cases. Further, the feature was integration tested with other features that were already stable in the language. All tests were then added to a global testing suite that could be automatically backtested at any time with a single command.

5.2 Style Guide

We generally followed the following guidelines while developing our compiler:

- All local variables are snake case, all AST types are camel case.
- Two spaces per indent.
- Generally stick to 80 char lines.
- Try to keep variable declarations at the top of the file.
- Misc functions for semantic checking and code generation stored in util.ml.
- Large code blocks were proceeded with multiline Ocaml comments explaining the following code.

5.3 Timeline

5.3.1 Planned Timeline

Below is the projected milestone timeline for our project, roughly laid out at the beginning of development. Because development was primarily feature oriented instead of component oriented, milestones represent features that were expected to have been completed by the given date.

Timeline

Date	Milestone
Sept. 26	Project Proposal
Oct. 16	Language Reference Manual
Nov. 8	Hello World (print, floats, strings)
Nov. 31	Structs, Function Pointers, Matrices
Dec 8	DEEP library abstraction, MNIST
Dec 18	Testing, Debugging complete
Dec 20	Final report complete

5.3.2 Project Log

Below is the actual timeline for our project. Again, because development was primarily feature oriented, milestones represent features that were completed by the given date. Further, because more features were added to the initial specification, the project log contains additional milestones that were not present in the planned timeline.

Timeline

Date	Milestone
Sep. 26	Project Proposal
Oct. 16	Language Reference Manual
Nov. 1	Project scrapped, restarted from scratch
Nov. 8	Hello World (print, floats, strings, C externals)
Nov. 25	Structs completed
Dec. 4	Matrices completed
Dec. 7	Arrays completed
Dec. 13	Function pointers completed
Dec. 14	Pipes and struct method distpatch, Math library completed
Dec. 15	MNIST completed
Dec. 18	DEEP library, testing and debugging complete
Dec. 20	Final report complete

5.4 Roles and Responsibilities

As previously mentioned, team responsibilities were assigned by feature. Each team member was responsible for front to back development of each feature. Thus, each team member touched all parts of the compiler and testing suites.

Team Responsibilities

Member	Responsibilities
Andrew Aday	Structs, Arrays, MNIST
Amol Kapoor	Extern, Function Pointers, Final Report
Jonathan Zhang	Eigen Matrix Linking, Native Matrix Features

5.5 Software Development Environment

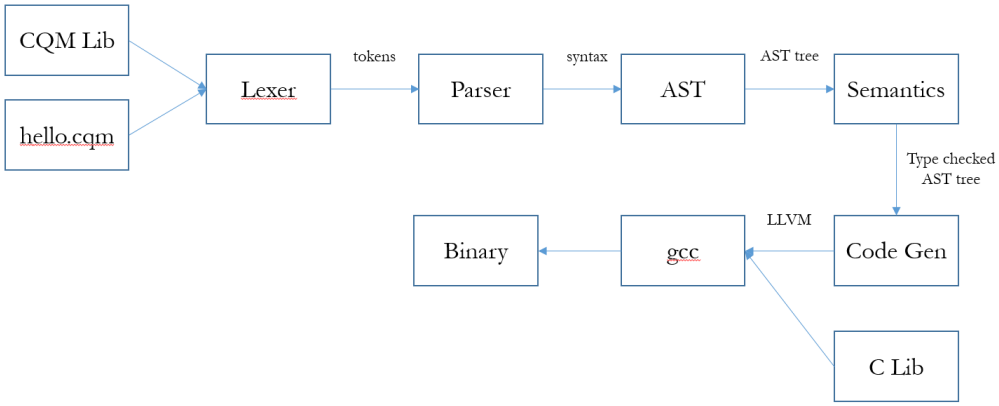
We used the following programming and development environment:

- Libraries and Languages: Ocaml version 4.05, including Ocamlyacc version 4.05 and Ocammllex version 4.05 extensions. LLVM Ocaml version 5.0. gcc version varies: version 9.0 and version 7.3 were both used.
- Software: Development was done on varying coding environments, including Atom, SublimeText, and Vim.
- OS: Development was done on OSX 10.13 and on Ubuntu 16.04.

6 Architectural Design

6.1 The Compiler

The architecture of the C? compiler consists of the following major components: CQM Libraries, Lexer, Parser, Semantic Checker, Code Generator, and C libraries. The architecture is shown as a system block diagram below.



The high level C? to LLVM compiler program `cqm.ml` calls each of the above components sequentially. Our C? compiler `compile.sh` calls the compiler made by `cqm.ml` and runs the entire compiler pipeline. First, input C? code is combined with any libraries (`.cqm`) files stored in the `/lib/` folder. This combined file is passed to the lexer and parser, generating an AST structure. The AST tree is passed to the semantic checker, which checks types and ensures semantic meaning. If the AST tree contains no type errors, the tree is passed to the code generation module, which converts the input into LLVM. The LLVM code is passed to gcc, which links to the necessary C libraries. gcc outputs an executable file.

6.1.1 Scanner

The scanner simply takes an input C? program and generates tokens to identify keywords, operators, and the various other language conventions described in the LRM.

6.1.2 Parser

The parser evaluates the tokens generated by the lexer and creates an Abstract Syntax Tree (AST) by specifying precedence and matching our recognized tokens with AST nodes.

6.1.3 Semantics

The semantics checker runs through the AST and mainly checks for proper typing. This process is especially important in collection based types such as arrays and matrices and user-defined types such as structs and function pointers. Our semantic checker does not return a SAST since it never directly modifies the base AST; rather it will throw an error if it encounters an illegal usage.

6.1.4 Code Generator

The code generator, takes a semantically checked AST and builds the LLVM equivalent of the language. The program simply walks through a post-order traversal and generates the appropriate LLVM call for each specific node in the AST.

6.2 Libraries

The main library used by our language is the popular Eigen[4] Linear Algebra library that provides common matrix, vector and other operations in C++. Owing to the large size of Eigen, we only take a small subset of the library and wrap the underlying matrix types and operations with a C library. The resulting compiled object file is then linked with gcc during the linking step. We also extended several useful functions from the standard math library in C++ using a similar method.

6.3 A Note on Labor Division

All components were built with input from all team members; each team member was responsible for specific features end-to-end, from syntax to testing.

[4] http://eigen.tuxfamily.org/

7 Test Plan

Features were developed independently in separate branches. For each feature, white box tests were developed that tested expected working and failing inputs. After the feature passed individual unit tests, multiple integration tests were built to test how the feature played with other parts of the language. The MNIST demo doubles as a system test. In total there are 123 different tests.

7.1 Testing Suites

All tests are stored in the `/test/` folder. Tests are split into a fail suite and a test suite based on name.

Any fail test follows the naming pattern `fail-*.cqm`. These tests have an expected output stored in `<FILENAME>.err`, where `FILENAME` is the same file as the original fail test.

Any normal test follows the naming pattern `test-*.cqm`. These tests have an expected output stored in `<FILENAME>.out`, where `FILENAME` is the same file as the original normal test.

7.2 Automation

Testing automation is based on the Micro-C testing suite. The `testall.sh` script compiles and runs all *.cqm files in the `/test/` folder and compares the output of the file to the corresponding *.err or *.out file of the same name. Any compilation errors or differences in output are passed to stdout.

7.3 Division of Labor

All team members contributed to testing development. Individual team members were responsible for unit testing the features that were assigned to them, and for coordinating with other teammates to build out the appropriate integration tests.

7.4 Example Input-Output

Original CQM file for Struct Array Matrix test.

```
1  struct foo {
2    fmatrix[] fms;
3    int[] a;
4  }
5
6  int main()
7  {
```

```
  struct foo foo;
  fmatrix fm;

  foo = make(struct foo);
  foo.a = make(int,5);
  foo.fms = make(fmatrix, 1);
  fm = init_fmat_identity(3,3);
  print_mat(fm);
  foo.fms[0] = init_fmat_identity(4,4);
  print_mat(foo.fms[0]);
  foo.fms[0] = fm;
  print_mat(foo.fms[0]);
  fm[0,0] = 3.14;
  print_mat(foo.fms[0]);

  return 0;
}
```

Output LLVM for Struct Array Matrix test.

```
; ModuleID = 'MicroC'
source_filename = "MicroC"

%struct.foo = type { i32**, i32* }

@__empty_string = global [1 x i8] zeroinitializer
@str = private unnamed_addr constant [19 x i8] c"rows: %d cols: %d\0A\00"
@str.1 = private unnamed_addr constant [2 x i8] c"\0A\00"
@str.2 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@str.3 = private unnamed_addr constant [4 x i8] c"%f\0A\00"
@str.4 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@str.5 = private unnamed_addr constant [4 x i8] c"%d\0A\00"

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

declare i32 @time(i32*, ...)

declare i32 @memcpy(i8*, i8*, i32)

declare i32* @init_fmat_literal(double*, i32, i32, ...)

declare i32* @negate(i32*)

declare i32* @transpose(i32*)

```

```
26  declare i32* @smeq(i32*, double)
27
28  declare i32* @sm_div(i32*, double)
29
30  declare i32* @sm_mult(i32*, double)
31
32  declare i32* @sm_sub(i32*, double, i32)
33
34  declare i32* @sm_add(i32*, double)
35
36  declare i32* @dot(i32*, i32*)
37
38  declare i32* @mm_div(i32*, i32*)
39
40  declare i32* @mm_mult(i32*, i32*)
41
42  declare i32* @mm_sub(i32*, i32*)
43
44  declare i32* @mm_add(i32*, i32*)
45
46  declare i32* @map(i32*, double (double)*)
47
48  declare void @del_mat(i32*)
49
50  declare i32* @copy(i32*)
51
52  declare i32* @arr_to_fmat(double*, i32, i32)
53
54  declare i32* @init_fmat_identity(i32, i32)
55
56  declare i32* @init_fmat_const(double, i32, i32)
57
58  declare i32* @init_fmat_zero(i32, i32)
59
60  declare i32 @cols(i32*)
61
62  declare i32 @rows(i32*)
63
64  declare double @mat_index_assign(i32*, i32, i32, double)
65
66  declare double @mat_index(i32*, i32, i32)
67
68  declare void @print_mat(i32*)
69
70  declare void @flush()
71
72  define i32 @main() {
```

```
entry:
  %foo = alloca %struct.foo*
  store %struct.foo* null, %struct.foo** %foo
  %fm = alloca i32*
  store i32* null, i32** %fm
  %malloccall = tail call i8* @malloc(i32 trunc (i64 mul nuw (i64 ptrtoint
   ↪  (i1** getelementptr (i1*, i1** null, i32 1) to i64), i64 2) to i32))
  %make_struct = bitcast i8* %malloccall to %struct.foo*
  store %struct.foo* %make_struct, %struct.foo** %foo
  %make_array = tail call i8* @malloc(i32 mul (i32 add (i32 mul (i32 ptrtoint
   ↪  (i32* getelementptr (i32, i32* null, i32 1) to i32), i32 5), i32 12),
   ↪  i32 ptrtoint (i8* getelementptr (i8, i8* null, i32 1) to i32)))
  %body_ptr = getelementptr i8, i8* %make_array, i8 12
  %i32_ptr_t = bitcast i8* %body_ptr to i32*
  %meta_ptr = getelementptr i32, i32* %i32_ptr_t, i32 -3
  store i32 ptrtoint (i32* getelementptr (i32, i32* null, i32 1) to i32), i32*
   ↪  %meta_ptr
  %i32_ptr_t2 = bitcast i8* %body_ptr to i32*
  %meta_ptr3 = getelementptr i32, i32* %i32_ptr_t2, i32 -2
  store i32 add (i32 mul (i32 ptrtoint (i32* getelementptr (i32, i32* null, i32
   ↪  1) to i32), i32 5), i32 12), i32* %meta_ptr3
  %i32_ptr_t4 = bitcast i8* %body_ptr to i32*
  %meta_ptr5 = getelementptr i32, i32* %i32_ptr_t4, i32 -1
  store i32 5, i32* %meta_ptr5
  %make_array_ptr = bitcast i8* %body_ptr to i32*
  %struct.foo = load %struct.foo*, %struct.foo** %foo
  %foo.a = getelementptr inbounds %struct.foo, %struct.foo* %struct.foo, i32 0,
   ↪  i32 1
  store i32* %make_array_ptr, i32** %foo.a
  %make_array7 = tail call i8* @malloc(i32 mul (i32 add (i32 ptrtoint (i1**
   ↪  getelementptr (i1*, i1** null, i32 1) to i32), i32 12), i32 ptrtoint
   ↪  (i8* getelementptr (i8, i8* null, i32 1) to i32)))
  %body_ptr8 = getelementptr i8, i8* %make_array7, i8 12
  %i32_ptr_t9 = bitcast i8* %body_ptr8 to i32*
  %meta_ptr10 = getelementptr i32, i32* %i32_ptr_t9, i32 -3
  store i32 ptrtoint (i1** getelementptr (i1*, i1** null, i32 1) to i32), i32*
   ↪  %meta_ptr10
  %i32_ptr_t11 = bitcast i8* %body_ptr8 to i32*
  %meta_ptr12 = getelementptr i32, i32* %i32_ptr_t11, i32 -2
  store i32 add (i32 ptrtoint (i1** getelementptr (i1*, i1** null, i32 1) to
   ↪  i32), i32 12), i32* %meta_ptr12
  %i32_ptr_t13 = bitcast i8* %body_ptr8 to i32*
  %meta_ptr14 = getelementptr i32, i32* %i32_ptr_t13, i32 -1
  store i32 1, i32* %meta_ptr14
  %make_array_ptr15 = bitcast i8* %body_ptr8 to i32**
  %struct.foo16 = load %struct.foo*, %struct.foo** %foo
  %foo.fms = getelementptr inbounds %struct.foo, %struct.foo* %struct.foo16,
   ↪  i32 0, i32 0
```

```
  store i32** %make_array_ptr15, i32*** %foo.fms
  %init_fmat_identity_result = call i32* @init_fmat_identity(i32 3, i32 3)
  store i32* %init_fmat_identity_result, i32** %fm
  %fm17 = load i32*, i32** %fm
  call void @print_mat(i32* %fm17)
  %struct.foo18 = load %struct.foo*, %struct.foo** %foo
  %foo.fms19 = getelementptr inbounds %struct.foo, %struct.foo* %struct.foo18,
   ↪  i32 0, i32 0
  %foo.fms_load = load i32**, i32*** %foo.fms19
  %init_fmat_identity_result20 = call i32* @init_fmat_identity(i32 4, i32 4)
  %"foo.fms[]" = getelementptr inbounds i32*, i32** %foo.fms_load, i32 0
  store i32* %init_fmat_identity_result20, i32** %"foo.fms[]"
  %struct.foo21 = load %struct.foo*, %struct.foo** %foo
  %foo.fms22 = getelementptr inbounds %struct.foo, %struct.foo* %struct.foo21,
   ↪  i32 0, i32 0
  %foo.fms_load23 = load i32**, i32*** %foo.fms22
  %"foo.fms[]24" = getelementptr inbounds i32*, i32** %foo.fms_load23, i32 0
  %"foo.fms[]_load" = load i32*, i32** %"foo.fms[]24"
  call void @print_mat(i32* %"foo.fms[]_load")
  %struct.foo25 = load %struct.foo*, %struct.foo** %foo
  %foo.fms26 = getelementptr inbounds %struct.foo, %struct.foo* %struct.foo25,
   ↪  i32 0, i32 0
  %foo.fms_load27 = load i32**, i32*** %foo.fms26
  %fm28 = load i32*, i32** %fm
  %"foo.fms[]29" = getelementptr inbounds i32*, i32** %foo.fms_load27, i32 0
  store i32* %fm28, i32** %"foo.fms[]29"
  %struct.foo30 = load %struct.foo*, %struct.foo** %foo
  %foo.fms31 = getelementptr inbounds %struct.foo, %struct.foo* %struct.foo30,
   ↪  i32 0, i32 0
  %foo.fms_load32 = load i32**, i32*** %foo.fms31
  %"foo.fms[]33" = getelementptr inbounds i32*, i32** %foo.fms_load32, i32 0
  %"foo.fms[]_load34" = load i32*, i32** %"foo.fms[]33"
  call void @print_mat(i32* %"foo.fms[]_load34")
  %fm35 = load i32*, i32** %fm
  %mat_index_assign_res = call double @mat_index_assign(i32* %fm35, i32 0, i32
   ↪  0, double 3.140000e+00)
  %struct.foo36 = load %struct.foo*, %struct.foo** %foo
  %foo.fms37 = getelementptr inbounds %struct.foo, %struct.foo* %struct.foo36,
   ↪  i32 0, i32 0
  %foo.fms_load38 = load i32**, i32*** %foo.fms37
  %"foo.fms[]39" = getelementptr inbounds i32*, i32** %foo.fms_load38, i32 0
  %"foo.fms[]_load40" = load i32*, i32** %"foo.fms[]39"
  call void @print_mat(i32* %"foo.fms[]_load40")
  ret i32 0
}

define void @free_fmat_arr(i32** %arr) {
```

```
entry:
  %arr1 = alloca i32**
  store i32** %arr, i32*** %arr1
  %i = alloca i32
  store i32 0, i32* %i
  store i32 0, i32* %i
  br label %while

while:                                            ; preds = %while_body, %entry
  %i5 = load i32, i32* %i
  %arr6 = load i32**, i32*** %arr1
  %null = icmp eq i32** %arr6, null
  %i32_ptr_t = bitcast i32** %arr6 to i32*
  %meta_ptr = getelementptr i32, i32* %i32_ptr_t, i32 -1
  %meta_data = load i32, i32* %meta_ptr
  %len = select i1 %null, i32 0, i32 %meta_data
  %tmp7 = icmp slt i32 %i5, %len
  br i1 %tmp7, label %while_body, label %merge

while_body:                                       ; preds = %while
  %i2 = load i32, i32* %i
  %arr3 = load i32**, i32*** %arr1
  %"arr[  %i2 = load i32, i32* %i]" = getelementptr inbounds i32*, i32** %arr3,
   ↪  i32 %i2
  %"arr[  %i2 = load i32, i32* %i]_load" = load i32*, i32** %"arr[  %i2 = load
   ↪  i32, i32* %i]"
  call void @del_mat(i32* %"arr[  %i2 = load i32, i32* %i]_load")
  %i4 = load i32, i32* %i
  %tmp = add i32 %i4, 1
  store i32 %tmp, i32* %i
  br label %while

merge:                                            ; preds = %while
  %arr8 = load i32**, i32*** %arr1
  %body_ptr = bitcast i32** %arr8 to i8*
  %meta_ptr9 = getelementptr i8, i8* %body_ptr, i8 -12
  tail call void @free(i8* %meta_ptr9)
  ret void
}

define i32* @f_fmat(i32* %fm, double (double)* %f) {
entry:
  %fm1 = alloca i32*
  store i32* %fm, i32** %fm1
  %f2 = alloca double (double)*
  store double (double)* %f, double (double)** %f2
  %i = alloca i32
```

```
196   store i32 0, i32* %i
197   %j = alloca i32
198   store i32 0, i32* %j
199   %fm13 = alloca i32*
200   store i32* null, i32** %fm13
201   %fm4 = load i32*, i32** %fm1
202   %copy_result = call i32* @copy(i32* %fm4)
203   store i32* %copy_result, i32** %fm13
204   store i32 0, i32* %i
205   br label %while
206
207 while:                                            ; preds = %merge, %entry
208   %i19 = load i32, i32* %i
209   %fm20 = load i32*, i32** %fm1
210   %rows_result = call i32 @rows(i32* %fm20)
211   %tmp21 = icmp slt i32 %i19, %rows_result
212   br i1 %tmp21, label %while_body, label %merge22
213
214 while_body:                                       ; preds = %while
215   store i32 0, i32* %j
216   br label %while5
217
218 while5:                                           ; preds = %while_body6,
      ↪  %while_body
219   %j14 = load i32, i32* %j
220   %fm15 = load i32*, i32** %fm1
221   %cols_result = call i32 @cols(i32* %fm15)
222   %tmp16 = icmp slt i32 %j14, %cols_result
223   br i1 %tmp16, label %while_body6, label %merge
224
225 while_body6:                                      ; preds = %while5
226   %fm17 = load i32*, i32** %fm13
227   %i8 = load i32, i32* %i
228   %j9 = load i32, i32* %j
229   %fm10 = load i32*, i32** %fm1
230   %i11 = load i32, i32* %i
231   %j12 = load i32, i32* %j
232   %mat_index_res = call double @mat_index(i32* %fm10, i32 %i11, i32 %j12)
233   %load_fptr = load double (double)*, double (double)** %f2
234   %f_result = call double %load_fptr(double %mat_index_res)
235   %mat_index_assign_res = call double @mat_index_assign(i32* %fm17, i32 %i8,
      ↪  i32 %j9, double %f_result)
236   %j13 = load i32, i32* %j
237   %tmp = add i32 %j13, 1
238   store i32 %tmp, i32* %j
239   br label %while5
240
```

```
merge:                                            ; preds = %while5
  %i17 = load i32, i32* %i
  %tmp18 = add i32 %i17, 1
  store i32 %tmp18, i32* %i
  br label %while

merge22:                                          ; preds = %while
  %fm123 = load i32*, i32** %fm13
  ret i32* %fm123
}

define i32* @populate_fmat(i32* %fm, double ()* %f) {
entry:
  %fm1 = alloca i32*
  store i32* %fm, i32** %fm1
  %f2 = alloca double ()*
  store double ()* %f, double ()** %f2
  %i = alloca i32
  store i32 0, i32* %i
  %j = alloca i32
  store i32 0, i32* %j
  store i32 0, i32* %i
  br label %while

while:                                            ; preds = %merge, %entry
  %i14 = load i32, i32* %i
  %fm15 = load i32*, i32** %fm1
  %rows_result = call i32 @rows(i32* %fm15)
  %tmp16 = icmp slt i32 %i14, %rows_result
  br i1 %tmp16, label %while_body, label %merge17

while_body:                                       ; preds = %while
  store i32 0, i32* %j
  br label %while3

while3:                                           ; preds = %while_body4,
 ↪  %while_body
  %j9 = load i32, i32* %j
  %fm10 = load i32*, i32** %fm1
  %cols_result = call i32 @cols(i32* %fm10)
  %tmp11 = icmp slt i32 %j9, %cols_result
  br i1 %tmp11, label %while_body4, label %merge

while_body4:                                      ; preds = %while3
  %fm5 = load i32*, i32** %fm1
  %i6 = load i32, i32* %i
  %j7 = load i32, i32* %j
```

```
  %load_fptr = load double ()*, double ()** %f2
  %f_result = call double %load_fptr()
  %mat_index_assign_res = call double @mat_index_assign(i32* %fm5, i32 %i6, i32 %j7, double %f_result)
  %j8 = load i32, i32* %j
  %tmp = add i32 %j8, 1
  store i32 %tmp, i32* %j
  br label %while3

merge:                                            ; preds = %while3
  %i12 = load i32, i32* %i
  %tmp13 = add i32 %i12, 1
  store i32 %tmp13, i32* %i
  br label %while

merge17:                                          ; preds = %while
  %fm18 = load i32*, i32** %fm1
  ret i32* %fm18
}

define void @print_fmat_arr_dims(i32** %arr) {
entry:
  %arr1 = alloca i32**
  store i32** %arr, i32*** %arr1
  %i = alloca i32
  store i32 0, i32* %i
  store i32 0, i32* %i
  br label %while

while:                                            ; preds = %while_body, %entry
  %i7 = load i32, i32* %i
  %arr8 = load i32**, i32*** %arr1
  %null = icmp eq i32** %arr8, null
  %i32_ptr_t = bitcast i32** %arr8 to i32*
  %meta_ptr = getelementptr i32, i32* %i32_ptr_t, i32 -1
  %meta_data = load i32, i32* %meta_ptr
  %len = select i1 %null, i32 0, i32 %meta_data
  %tmp9 = icmp slt i32 %i7, %len
  br i1 %tmp9, label %while_body, label %merge

while_body:                                       ; preds = %while
  %i2 = load i32, i32* %i
  %arr3 = load i32**, i32*** %arr1
  %"arr[  %i2 = load i32, i32* %i]" = getelementptr inbounds i32*, i32** %arr3, i32 %i2
  %"arr[  %i2 = load i32, i32* %i]_load" = load i32*, i32** %"arr[  %i2 = load i32, i32* %i]"
```

```
  %rows_result = call i32 @rows(i32* %"arr[  %i2 = load i32, i32* %i]_load")
  %i4 = load i32, i32* %i
  %arr5 = load i32**, i32*** %arr1
  %"arr[  %i4 = load i32, i32* %i]" = getelementptr inbounds i32*, i32** %arr5, i32 %i4
  %"arr[  %i4 = load i32, i32* %i]_load" = load i32*, i32** %"arr[  %i4 = load i32, i32* %i]"
  %cols_result = call i32 @cols(i32* %"arr[  %i4 = load i32, i32* %i]_load")
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([19 x i8], [19 x i8]* @str, i32 0, i32 0), i32 %rows_result, i32 %cols_result)
  %i6 = load i32, i32* %i
  %tmp = add i32 %i6, 1
  store i32 %tmp, i32* %i
  br label %while

merge:                                            ; preds = %while
  ret void
}

define void @print_fmat_arr(i32** %arr) {
entry:
  %arr1 = alloca i32**
  store i32** %arr, i32*** %arr1
  %i = alloca i32
  store i32 0, i32* %i
  store i32 0, i32* %i
  br label %while

while:                                            ; preds = %while_body, %entry
  %i5 = load i32, i32* %i
  %arr6 = load i32**, i32*** %arr1
  %null = icmp eq i32** %arr6, null
  %i32_ptr_t = bitcast i32** %arr6 to i32*
  %meta_ptr = getelementptr i32, i32* %i32_ptr_t, i32 -1
  %meta_data = load i32, i32* %meta_ptr
  %len = select i1 %null, i32 0, i32 %meta_data
  %tmp7 = icmp slt i32 %i5, %len
  br i1 %tmp7, label %while_body, label %merge

while_body:                                       ; preds = %while
  %i2 = load i32, i32* %i
  %arr3 = load i32**, i32*** %arr1
  %"arr[  %i2 = load i32, i32* %i]" = getelementptr inbounds i32*, i32** %arr3, i32 %i2
  %"arr[  %i2 = load i32, i32* %i]_load" = load i32*, i32** %"arr[  %i2 = load i32, i32* %i]"
  call void @print_mat(i32* %"arr[  %i2 = load i32, i32* %i]_load")
```

```
373   %i4 = load i32, i32* %i
374   %tmp = add i32 %i4, 1
375   store i32 %tmp, i32* %i
376   br label %while
377 
378 merge:                                            ; preds = %while
379   ret void
380 }
381 
382 define void @print_line() {
383 entry:
384   %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([2 x i8],
      ↪  [2 x i8]* @str.1, i32 0, i32 0))
385   ret void
386 }
387 
388 define void @print_string(i8* %s) {
389 entry:
390   %s1 = alloca i8*
391   store i8* %s, i8** %s1
392   %s2 = load i8*, i8** %s1
393   %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
      ↪  [4 x i8]* @str.2, i32 0, i32 0), i8* %s2)
394   ret void
395 }
396 
397 define void @print_float(double %f) {
398 entry:
399   %f1 = alloca double
400   store double %f, double* %f1
401   %f2 = load double, double* %f1
402   %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
      ↪  [4 x i8]* @str.3, i32 0, i32 0), double %f2)
403   ret void
404 }
405 
406 define void @printb(i1 %b) {
407 entry:
408   %b1 = alloca i1
409   store i1 %b, i1* %b1
410   %b2 = load i1, i1* %b1
411   %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
      ↪  [4 x i8]* @str.4, i32 0, i32 0), i1 %b2)
412   ret void
413 }
414 
415 define void @print(i32 %i) {
```

```
entry:
  %i1 = alloca i32
  store i32 %i, i32* %i1
  %i2 = load i32, i32* %i1
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
   ↪  [4 x i8]* @str.5, i32 0, i32 0), i32 %i2)
  ret void
}

declare noalias i8* @malloc(i32)

declare void @free(i8*)
```

Original CQM file for Sieve of Eranthoses.

```
void sieve_of_eratosthenes(int n)
{
  bool[] prime;
  int: i, p;

  prime = make(bool, n);
  for (i = 0; i < len(prime) + 1; i = i + 1) {
    prime[i] = true;
  }

  p = 2;
  while (p * p <= n) {
    if (prime[p]) {
      for (i = 2*p; i < len(prime) + 1; i = i + p) {
        prime[i] = false;
      }
    }
    p = p + 1;
  }

  for (i = 2; i < n + 1; i = i + 1) {
    if (prime[i]) {
      print(i);
    }
  }
}

int main()
{
  int n;
```

```
  n = 100;

  sieve_of_eratosthenes(n);
}
```

Output LLVM for Sieve of Eranthoses.

```
; ModuleID = 'MicroC'
source_filename = "MicroC"

@__empty_string = global [1 x i8] zeroinitializer
@str = private unnamed_addr constant [2 x i8] c"\0A\00"
@str.1 = private unnamed_addr constant [4 x i8] c"%s\0A\00"
@str.2 = private unnamed_addr constant [4 x i8] c"%f\0A\00"
@str.3 = private unnamed_addr constant [4 x i8] c"%d\0A\00"
@str.4 = private unnamed_addr constant [4 x i8] c"%d\0A\00"

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

declare i32 @time(i32*, ...)

declare i32 @memcpy(i8*, i8*, i32)

declare i32* @init_fmat_literal(double*, i32, i32, ...)

declare void @flush()

define i32 @main() {
entry:
  %n = alloca i32
  store i32 0, i32* %n
  store i32 100, i32* %n
  %n1 = load i32, i32* %n
  call void @sieve_of_eratosthenes(i32 %n1)
  ret i32 0
}

define void @sieve_of_eratosthenes(i32 %n) {
entry:
  %n1 = alloca i32
  store i32 %n, i32* %n1
  %prime = alloca i1*
  store i1* null, i1** %prime
  %i = alloca i32
```

```
  store i32 0, i32* %i
  %p = alloca i32
  store i32 0, i32* %p
  %n2 = load i32, i32* %n1
  %body_sz = mul i32 ptrtoint (i1* getelementptr (i1, i1* null, i32 1) to i32),
   ↪  %n2
  %make_array_sz = add i32 %body_sz, 12
  %mallocsize = mul i32 %make_array_sz, ptrtoint (i8* getelementptr (i8, i8*
   ↪  null, i32 1) to i32)
  %make_array = tail call i8* @malloc(i32 %mallocsize)
  %body_ptr = getelementptr i8, i8* %make_array, i8 12
  %i32_ptr_t = bitcast i8* %body_ptr to i32*
  %meta_ptr = getelementptr i32, i32* %i32_ptr_t, i32 -3
  store i32 ptrtoint (i1* getelementptr (i1, i1* null, i32 1) to i32), i32*
   ↪  %meta_ptr
  %i32_ptr_t3 = bitcast i8* %body_ptr to i32*
  %meta_ptr4 = getelementptr i32, i32* %i32_ptr_t3, i32 -2
  store i32 %make_array_sz, i32* %meta_ptr4
  %i32_ptr_t5 = bitcast i8* %body_ptr to i32*
  %meta_ptr6 = getelementptr i32, i32* %i32_ptr_t5, i32 -1
  store i32 %n2, i32* %meta_ptr6
  %make_array_ptr = bitcast i8* %body_ptr to i1*
  store i1* %make_array_ptr, i1** %prime
  store i32 0, i32* %i
  br label %while

while:                                            ; preds = %while_body, %entry
  %i10 = load i32, i32* %i
  %prime11 = load i1*, i1** %prime
  %null = icmp eq i1* %prime11, null
  %i32_ptr_t12 = bitcast i1* %prime11 to i32*
  %meta_ptr13 = getelementptr i32, i32* %i32_ptr_t12, i32 -1
  %meta_data = load i32, i32* %meta_ptr13
  %len = select i1 %null, i32 0, i32 %meta_data
  %tmp14 = add i32 %len, 1
  %tmp15 = icmp slt i32 %i10, %tmp14
  br i1 %tmp15, label %while_body, label %merge

while_body:                                       ; preds = %while
  %i7 = load i32, i32* %i
  %prime8 = load i1*, i1** %prime
  %"prime[  %i7 = load i32, i32* %i]" = getelementptr inbounds i1, i1* %prime8,
   ↪  i32 %i7
  store i1 true, i1* %"prime[  %i7 = load i32, i32* %i]"
  %i9 = load i32, i32* %i
  %tmp = add i32 %i9, 1
  store i32 %tmp, i32* %i
```

```
  br label %while

merge:                                            ; preds = %while
  store i32 2, i32* %p
  br label %while16

while16:                                          ; preds = %merge20, %merge
  %p42 = load i32, i32* %p
  %p43 = load i32, i32* %p
  %tmp44 = mul i32 %p42, %p43
  %n45 = load i32, i32* %n1
  %tmp46 = icmp sle i32 %tmp44, %n45
  br i1 %tmp46, label %while_body17, label %merge47

while_body17:                                     ; preds = %while16
  %p18 = load i32, i32* %p
  %prime19 = load i1*, i1** %prime
  %"prime[  %p18 = load i32, i32* %p]" = getelementptr inbounds i1, i1*
   ↪  %prime19, i32 %p18
  %"prime[  %p18 = load i32, i32* %p]_load" = load i1, i1* %"prime[  %p18 =
   ↪  load i32, i32* %p]"
  br i1 %"prime[  %p18 = load i32, i32* %p]_load", label %then, label %else

merge20:                                          ; preds = %else, %merge39
  %p40 = load i32, i32* %p
  %tmp41 = add i32 %p40, 1
  store i32 %tmp41, i32* %p
  br label %while16

then:                                             ; preds = %while_body17
  %p21 = load i32, i32* %p
  %tmp22 = mul i32 2, %p21
  store i32 %tmp22, i32* %i
  br label %while23

while23:                                          ; preds = %while_body24,
 ↪  %then
  %i30 = load i32, i32* %i
  %prime31 = load i1*, i1** %prime
  %null32 = icmp eq i1* %prime31, null
  %i32_ptr_t33 = bitcast i1* %prime31 to i32*
  %meta_ptr34 = getelementptr i32, i32* %i32_ptr_t33, i32 -1
  %meta_data35 = load i32, i32* %meta_ptr34
  %len36 = select i1 %null32, i32 0, i32 %meta_data35
  %tmp37 = add i32 %len36, 1
  %tmp38 = icmp slt i32 %i30, %tmp37
  br i1 %tmp38, label %while_body24, label %merge39
```

```
125
126 while_body24:                                    ; preds = %while23
127   %i25 = load i32, i32* %i
128   %prime26 = load i1*, i1** %prime
129   %"prime[  %i25 = load i32, i32* %i]" = getelementptr inbounds i1, i1*
      ↪  %prime26, i32 %i25
130   store i1 false, i1* %"prime[  %i25 = load i32, i32* %i]"
131   %i27 = load i32, i32* %i
132   %p28 = load i32, i32* %p
133   %tmp29 = add i32 %i27, %p28
134   store i32 %tmp29, i32* %i
135   br label %while23
136
137 merge39:                                          ; preds = %while23
138   br label %merge20
139
140 else:                                             ; preds = %while_body17
141   br label %merge20
142
143 merge47:                                          ; preds = %while16
144   store i32 2, i32* %i
145   br label %while48
146
147 while48:                                         ; preds = %merge52, %merge47
148   %i58 = load i32, i32* %i
149   %n59 = load i32, i32* %n1
150   %tmp60 = add i32 %n59, 1
151   %tmp61 = icmp slt i32 %i58, %tmp60
152   br i1 %tmp61, label %while_body49, label %merge62
153
154 while_body49:                                     ; preds = %while48
155   %i50 = load i32, i32* %i
156   %prime51 = load i1*, i1** %prime
157   %"prime[  %i50 = load i32, i32* %i]" = getelementptr inbounds i1, i1*
      ↪  %prime51, i32 %i50
158   %"prime[  %i50 = load i32, i32* %i]_load" = load i1, i1* %"prime[  %i50 =
      ↪  load i32, i32* %i]"
159   br i1 %"prime[  %i50 = load i32, i32* %i]_load", label %then53, label %else55
160
161 merge52:                                          ; preds = %else55, %then53
162   %i56 = load i32, i32* %i
163   %tmp57 = add i32 %i56, 1
164   store i32 %tmp57, i32* %i
165   br label %while48
166
167 then53:                                           ; preds = %while_body49
168   %i54 = load i32, i32* %i
```

```
169   call void @print(i32 %i54)
170   br label %merge52
171
172 else55:                                           ; preds = %while_body49
173   br label %merge52
174
175 merge62:                                          ; preds = %while48
176   ret void
177 }
178
179 define void @print_line() {
180 entry:
181   %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([2 x i8],
      ↪  [2 x i8]* @str, i32 0, i32 0))
182   ret void
183 }
184
185 define void @print_string(i8* %s) {
186 entry:
187   %s1 = alloca i8*
188   store i8* %s, i8** %s1
189   %s2 = load i8*, i8** %s1
190   %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
      ↪  [4 x i8]* @str.1, i32 0, i32 0), i8* %s2)
191   ret void
192 }
193
194 define void @print_float(double %f) {
195 entry:
196   %f1 = alloca double
197   store double %f, double* %f1
198   %f2 = load double, double* %f1
199   %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
      ↪  [4 x i8]* @str.2, i32 0, i32 0), double %f2)
200   ret void
201 }
202
203 define void @printb(i1 %b) {
204 entry:
205   %b1 = alloca i1
206   store i1 %b, i1* %b1
207   %b2 = load i1, i1* %b1
208   %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
      ↪  [4 x i8]* @str.3, i32 0, i32 0), i1 %b2)
209   ret void
210 }
```

```
211
212 define void @print(i32 %i) {
213 entry:
214   %i1 = alloca i32
215   store i32 %i, i32* %i1
216   %i2 = load i32, i32* %i1
217   %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8],
      ↪   [4 x i8]* @str.4, i32 0, i32 0), i32 %i2)
218   ret void
219 }
220
221 declare noalias i8* @malloc(i32)
```

8 Lessons Learned

8.1 Andrew Aday

PLT gave me the chance to learn a lot about Machine Learning, a field that I had no prior experience in. My team members were both pretty experienced in ML, and we ended up deciding to focus on an ML-based language. In order to understand both the direction of our language and our final implementation, I had to quickly get at least a basic knowledge of the core principles in ML. I ended up learning not just OCaml (and compiler design) but also the principles of deep learning. I suggest that students be creative with their languages and, if they have the support to do so, consider taking risks with learning something new.

8.2 Amol Kapoor

Sometimes it is easier and faster to start from scratch. The first iteration of C? was as Inception, a layers-based machine-learning-only library. After struggling with LLVM and OCaml for close to a month, we realized that our plan of action was wrong - instead of building a native machine learning language, we should have built a powerful general language that could support a machine learning library. Restarting the project proved to be far more fruitful than sticking with what ultimately would have been a long uphill slog. My advice: be comfortable with changing plans, even if the new plan is radically different.

8.3 Jonathan Zhang

It was a really bad idea to take so many hard classes in the same semester. PLT was the cherry on top of ML, Algos, Modern Algebra, and CC - not exactly a light workload by any means. Besides losing a lot of sleep, I felt that I missed out on a lot of the fun of building your own language. Instead of flexing any creativity, I had to do the bare minimum and spend time on other classes. I would advise students to be smart about what classes they are taking along with PLT. PLT is a lot of fun, but only if you have the time to do it.

9 Appendix

9.1 Shell scripts

compile.sh

```
LLI="lli"
LLC="llc"
CC="cc"
CQM="./cqm.native"

Run() {
    echo $* 1>&2
    eval $* || {
        SignalError "$1 failed on $*"
        return 1
    }
}

Compile() {
  basename=`echo $1 | sed 's/.*\\///
                           s/.cqm//'`
  echo ${basename}
  # Run "cat" "lib/*.cqm" "£1" "|" "£CQM" ">" "£{basename}.ll" &&
  Run "cat" "lib/*.cqm" "$1" "|" "$CQM" ">" "${basename}.ll" &&
  Run "$LLC" "${basename}.ll" ">" "${basename}.s" &&
  Run "$CC" "-o" "${basename}.exe" "${basename}.s" "printbig.o" "eigen_mnist.o"
   ↪  "-Llib/src" "-leigentest" "-lm"
}

Compile $1
```

testall.sh

```
#!/bin/sh

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

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

# Path to the LLVM compiler
```

```
13  LLC="llc"
14
15  # Path to the C compiler
16  CC="cc"
17
18  # Path to the cqm compiler.  Usually "./cqm.native"
19  # Try "_build/cqm.native" if ocamlbuild was unable to create a symbolic link.
20  CQM="./cqm.native"
21  #cqm="_build/cqm.native"
22
23
24  # Set time limit for all operations
25  ulimit -t 30
26
27  globallog=testall.log
28  rm -f $globallog
29  error=0
30  globalerror=0
31
32  keep=0
33
34  Usage() {
35      echo "Usage: testall.sh [options] [.cqm files]"
36      echo "-k    Keep intermediate files"
37      echo "-h    Print this help"
38      exit 1
39  }
40
41  SignalError() {
42      if [ $error -eq 0 ] ; then
43          echo "FAILED"
44          error=1
45      fi
46      echo "  $1"
47  }
48
49  # Compare <outfile> <reffile> <difffile>
50  # Compares the outfile with reffile.  Differences, if any, written to difffile
51  Compare() {
52      generatedfiles="$generatedfiles $3"
53      echo diff -b $1 $2 ">" $3 1>&2
54      diff -b "$1" "$2" > "$3" 2>&1 || {
55          SignalError "$1 differs"
56          echo "FAILED $1 differs from $2" 1>&2
57      }
58  }
59
```

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

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

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

    echo -n "$basename..."

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

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.s
     ↪   ${basename}.exe ${basename}.out" &&
    Run "cat" "lib/*.cqm" "$1" "|" "$CQM" ">" "${basename}.ll" &&
    Run "$LLC" "${basename}.ll" ">" "${basename}.s" &&
    Run "$CC" "-O3" "-o" "${basename}.exe" "${basename}.s" "printbig.o"
     ↪   "eigen_mnist.o" "-L lib/src" "-leigentest" "-lm" &&
    Run "./${basename}.exe" > "${basename}.out" &&
    Compare ${basename}.out ${reffile}.out ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
```

```
105         if [ $keep -eq 0 ] ; then
106             rm -f $generatedfiles
107         fi
108         echo "OK"
109         echo "###### SUCCESS" 1>&2
110     else
111         echo "###### FAILED" 1>&2
112         globalerror=$error
113     fi
114 }
115 
116 CheckFail() {
117     error=0
118     basename=`echo $1 | sed 's/.*\\///
119                               s/.cqm//'`
120     reffile=`echo $1 | sed 's/.cqm$//'`
121     basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."
122 
123     echo -n "$basename..."
124 
125     echo 1>&2
126     echo "###### Testing $basename" 1>&2
127 
128     generatedfiles=""
129 
130     generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
131     RunFail "cat" "lib/*.cqm" "$1" "|" "$CQM" "2>" "${basename}.err" ">>"
      ↪  $globallog &&
132     Compare ${basename}.err ${reffile}.err ${basename}.diff
133 
134     # Report the status and clean up the generated files
135 
136     if [ $error -eq 0 ] ; then
137         if [ $keep -eq 0 ] ; then
138             rm -f $generatedfiles
139         fi
140         echo "OK"
141         echo "###### SUCCESS" 1>&2
142     else
143         echo "###### FAILED" 1>&2
144         globalerror=$error
145     fi
146 }
147 
148 while getopts kdpsh c; do
149     case $c in
150         k) # Keep intermediate files
```

```
            keep=1
            ;;
        h) # Help
            Usage
            ;;
    esac
done

shift `expr $OPTIND - 1`

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

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

if [ ! -f printbig.o ]
then
    echo "Could not find printbig.o"
    echo "Try \"make printbig.o\""
    exit 1
fi

# if [ ! -f ./lib/src/libeigentest.so ]
# then
#     echo "Could not find eigen lib"
#     echo "Make sure libeigentest.so exists in ."
#     exit 1
# fi

if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/test-*.cqm tests/fail-*.cqm"
fi

for file in $files
do
    case $file in
        *test-*)
            Check $file 2>> $globallog
            ;;
        *fail-*)
```

```
            CheckFail $file 2>> $globallog
            ;;
        *)
            echo "unknown file type $file"
            globalerror=1
            ;;
    esac
done

exit $globalerror
```

compile (mnist)

```
LLI="lli"
LLC="llc"
CC="cc"
CQM="../cqm.native"

Run() {
    echo $* 1>&2
    eval $* || {
            SignalError "$1 failed on $*"
            return 1
    }
}

Compile() {
  basename=`echo $1 | sed 's/.*\\///
                           s/.cqm//'`
  echo ${basename}
  Run "cat" "../lib/*.cqm" "$1" "|" "$CQM" ">" "${basename}.ll" &&
  Run "$LLC" "${basename}.ll" ">" "${basename}.s" &&
  Run "$CC" "-O3" "-o" "${basename}" "${basename}.s" "eigen_mnist.o" "-L."
   ↪  "-leigentest" "-lm"
}

Compile $1
```

prelude (mnist)

```
if [ -f libeigentest.so ]; then
  rm libeigentest.so
fi
```

```
4
5  if [ -f eigen_mnist.o ]; then
6    rm eigen_mnist.o
7  fi
8
9  ln -s ../lib/src/libeigentest.so libeigentest.so
10 cp ../lib/src/eigen_mnist.o eigen_mnist.o
```

9.2 Compiler files

cqm.ml

```
1  (* Top-level of the MicroC compiler: scan & parse the input,
2     check the resulting AST, generate LLVM IR, and dump the module *)
3
4  module StringMap = Map.Make(String)
5
6  type action = Ast | LLVM_IR | Compile
7
8  let _ =
9    let action = ref Compile in
10   let set_action a () = action := a in
11   let speclist = [
12     ("-a", Arg.Unit (set_action Ast), "Print the SAST");
13     ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
14     ("-c", Arg.Unit (set_action Compile),
15       "Check and print the generated LLVM IR (default)");
16   ] in
17   let usage_msg = "usage: ./microc.native [-a|-l|-c] [file.cqm]" in
18   let channel = ref stdin in
19   Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;
20   let lexbuf = Lexing.from_channel !channel in
21   let ast =
22     try
23       Parser.program Scanner.token lexbuf
24     with exn ->
25       (
26         let curr = lexbuf.Lexing.lex_curr_p in
27         let line = curr.Lexing.pos_lnum in
28         let cnum = curr.Lexing.pos_cnum - curr.Lexing.pos_bol in
29         let tok = Lexing.lexeme lexbuf in
30         let failure_string = Scanf.unescaped(
31           "Exception: " ^ Printexc.to_string exn ^ "\n" ^
32           "Line number: " ^ (string_of_int line) ^ "\n" ^
```

```
          "Character: " ^ (string_of_int cnum) ^  "\n" ^
          "Token: " ^ tok
        ) in
        raise (Failure failure_string)
      )
  in
  Semant.check ast;
  (* ast; *)
  match !action with
    Ast -> print_string (Util.string_of_program ast)
  | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate ast))
  | Compile -> let m = Codegen.translate ast in
    Llvm_analysis.assert_valid_module m;
    print_string (Llvm.string_of_llmodule m)
```

scanner.mll

```
(* Ocamllex scanner for MicroC *)

{ open Parser }

let esc    = '\\' ['\\' ''' '"' 'n' 'r' 't']
let ascii  = ([' '-'!' '#'-'[' ']'-'~'])
let id = ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*

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

```
29  | "=>"     { PIPE }
30  | "**"     { POW }
31  | '%'      { MOD }
32  | '^'      { MATTRANS }
33  | ".."     { DOT }
34  | ':'      { SLICE }
35  | "=="     { EQ }
36  | "!="     { NEQ }
37  | '<'      { LT }
38  | "<="     { LEQ }
39  | ">"      { GT }
40  | ">="     { GEQ }
41  | "&&"     { AND }
42  | "||"     { OR }
43  | "!"      { NOT }
44  (*----------------------------------CONTROL----------------------------------*)
45  | "if"     { IF }
46  | "else"   { ELSE }
47  | "for"    { FOR }
48  | "while"  { WHILE }
49  | "return" { RETURN }
50  | "extern" { EXTERN }
51  | "make"   { MAKE }
52  (*-----------------------------------TYPES-----------------------------------*)
53  | "int"                  { INT }
54  | "bool"                 { BOOL }
55  | "void"                 { VOID }
56  | "float"          { FLOAT }
57  | "string"          { STRING }
58  | "imatrix"          { IMATRIX }
59  | "fmatrix"          { FMATRIX }
60  | "struct"               { STRUCT }
61  | "fp"               { FPTR }
62  (*----------------------------------LITERALS---------------------------------*)
63  | "true"   { TRUE }
64  | "false"  { FALSE }
65  | "NULL"   { NULL }
66  | ['0'-'9']+ as lxm                                          {
    ↪  INTLIT(int_of_string lxm) }
67  | (['0'-'9']+'.'['0'-'9']* | ['0'-'9']*'.'['0'-'9']+) as lxm  {
    ↪  FLOATLIT(float_of_string lxm) }
68  | '"' ((ascii | esc)* as s)'"'                               { STRINGLIT(s) }
69  | id as lxm                                                  { ID(lxm) }
70  | eof                                                        { EOF }
71  | _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }
72
73  and comment = parse
```

```
  "*/" { token lexbuf }
| _    { comment lexbuf }

and inline_comment = parse
  ['\n'] { token lexbuf }
| _    { inline_comment lexbuf }
```

parser.mly

```
/* Ocamlyacc parser for Onion */

%{
open Ast
%}


%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA PERIOD LBRACK RBRACK BAR
%token PLUS MINUS TIMES DIVIDE POW ASSIGN PIPE MOD MATTRANS DOT SLICE
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR NOT NULL
%token RETURN IF ELSE FOR WHILE EXTERN MAKE
%token INT BOOL VOID FLOAT STRING IMATRIX FMATRIX STRUCT FPTR
%token <int> INTLIT
%token <string> STRINGLIT
%token <float> FLOATLIT
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE

%right ASSIGN
%left PIPE
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left SLICE
%left PLUS MINUS
%left TIMES DIVIDE MOD DOT
%left POW
%right NOT NEG
%left MATTRANS

%start program
%type <Ast.program> program
```

```
37
38  %%
39
40  program:
41    decls EOF { $1 }
42
43  decls:
44     /* nothing */  {{ global_vars = []; functions = []; structs = []; }}
45   | decls vdecl    {{
46                       global_vars = $2 :: $1.global_vars;
47                       functions = $1.functions;
48                       structs = $1.structs;
49                     }}
50   | decls fdecl    {{
51                       global_vars = $1.global_vars;
52                       functions = $2 :: $1.functions;
53                       structs = $1.structs;
54                     }}
55   | decls str_decl {{
56                       global_vars = $1.global_vars;
57                       functions = $1.functions;
58                       structs = List.rev ($2 :: (List.rev ($1.structs)));
59                     }}
60   | decls str_mthd_decl {{
61                           global_vars = $1.global_vars;
62                           functions = $2 :: $1.functions;
63                           structs = $1.structs;
64                         }}
65  str_mthd_decl:
66    LBRACK struct_name ID RBRACK ID LPAREN formals_opt RPAREN typ LBRACE
     ↪  vdecl_list stmt_list RBRACE
67      {{
68        typ         = $9;
69              fname       = "__" ^ $2 ^ "_" ^ $5;
70              formals     = (StructType($2), $3) :: $7;
71              locals      = List.rev $11;
72        body        = List.rev $12;
73        location    = Local;
74      }}
75
76  struct_name:
77    STRUCT ID { $2 }
78
79  str_decl:
80    STRUCT ID LBRACE vdecl_list RBRACE
81    {{
82      name = $2;
```

```
    members = List.rev $4;
  }}

fdecl:
   typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
      { { typ         = $1;
                fname       = $2;
                formals     = $4;
                locals      = List.rev $7;
          body        = List.rev $8;
          location    = Local; } }
 | EXTERN typ ID LPAREN formals_opt RPAREN SEMI
      { { typ         = $2;
          fname       = $3;
          formals     = $5;
          locals      = [];
          body        = [];
          location    = External; } }

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

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

/*============================Type
 ↪  Parsing================================*/
typ:
    primitive_type {PrimitiveType($1)}
  | struct_type    {$1}
  | array_type     {$1}
  | fptr_type      {$1}

primitive_type:
    INT     { Int }
  | FLOAT   { Float }
  | STRING  { String }
  | BOOL    { Bool }
  | VOID    { Void }
  | IMATRIX { Imatrix }
  | FMATRIX { Fmatrix }

struct_type:
    STRUCT ID { StructType($2) }

```

```
array_type:
    typ LBRACK RBRACK { ArrayType($1) }

fptr_type:
    FPTR LPAREN typ_list RPAREN { FptrType(List.rev $3) }

typ_list:
    typ                   { [$1] }
  | typ_list COMMA typ { $3 :: $1 }

vdecl_list:
    /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }
  | vdecl_list multi_vdecl { $2 @ $1 }

vdecl:
    typ ID SEMI                     { ($1, $2) }

multi_vdecl:
    typ SLICE id_list SEMI      { List.map (fun (id) -> ($1, id)) $3 }

id_list:
    ID               { [$1] }
  | id_list COMMA ID  { $3 :: $1 }

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

stmt:
    expr SEMI                                 { Expr $1 }
  | RETURN SEMI                               { Return Noexpr }
  | RETURN expr SEMI                          { Return $2 }
  | LBRACE stmt_list RBRACE                   { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE   { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt      { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
                                              { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt             { While($3, $5) }

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

expr:
    INTLIT           { IntLit($1) }
  | FLOATLIT         { FloatLit($1) }
```

```
176   | STRINGLIT        { StringLit($1) }
177   | TRUE             { BoolLit(true) }
178   | FALSE            { BoolLit(false) }
179   | ID               { Id($1) }
180   | NULL             { Null }
181   | expr PLUS   expr { Binop($1, Add,     $3) }
182   | expr MINUS  expr { Binop($1, Sub,     $3) }
183   | expr TIMES  expr { Binop($1, Mult,    $3) }
184   | expr DIVIDE expr { Binop($1, Div,     $3) }
185   | expr POW    expr { Binop($1, Pow,     $3) }
186   | expr MOD    expr { Binop($1, Mod,     $3) }
187   | expr EQ     expr { Binop($1, Equal,   $3) }
188   | expr NEQ    expr { Binop($1, Neq,     $3) }
189   | expr LT     expr { Binop($1, Less,    $3) }
190   | expr LEQ    expr { Binop($1, Leq,     $3) }
191   | expr GT     expr { Binop($1, Greater, $3) }
192   | expr GEQ    expr { Binop($1, Geq,   $3) }
193   | expr AND    expr { Binop($1, And,   $3) }
194   | expr OR     expr { Binop($1, Or,    $3) }
195   | expr DOT expr    { Binop($1, Dot, $3)}
196   | expr MATTRANS    { Unop(Transpose, $1) }
197   | MINUS expr %prec NEG                                  { Unop(Neg, $2) }
198   | NOT expr                                              { Unop(Not, $2) }
199   | ID ASSIGN expr                                        { Assign($1, $3) }
200   | MAKE LPAREN typ RPAREN                                { MakeStruct($3) }
201   | MAKE LPAREN typ COMMA expr RPAREN                     { MakeArray($3, $5) }
202   | ID LPAREN actuals_opt RPAREN                          { Call($1, $3) }
203   | LPAREN expr RPAREN                                    { $2 }
204   | ID PERIOD ID                                          { StructAccess($1,
      ↪  $3) }
205   | ID PERIOD ID ASSIGN expr                              { StructAssign($1,
      ↪  $3, $5) }
206   | ID LBRACK expr RBRACK                                 { ArrayAccess($1, $3)
      ↪  }
207   | LBRACK rows RBRACK                                    { MatLit(List.rev $2)
      ↪  }
208   | ID LBRACK expr RBRACK ASSIGN expr                     { ArrayAssign($1, $3,
      ↪  $6) }
209   | LPAREN array_type RPAREN LBRACE actuals_opt RBRACE    { ArrayLit($2, $5) }
210   | expr PIPE expr                                        { Pipe($1, $3) }
211   | ID PERIOD ID LPAREN actuals_opt RPAREN                { Dispatch($1, $3,
      ↪  $5) }
212   | ID LBRACK expr COMMA expr RBRACK                      { MatIndex($1, $3,
      ↪  $5) }
213   | ID LBRACK expr COMMA expr RBRACK ASSIGN expr          { MatIndexAssign($1,
      ↪  $3, $5, $8) }
214   | ID PERIOD ID LBRACK expr RBRACK                       {
      ↪  StructArrayAccess($1, $3, $5) }
```

```
  | ID PERIOD ID LBRACK expr RBRACK ASSIGN expr          {
   ↪  StructArrayAssign($1, $3, $5, $8) }
  /*| LPAREN struct_type RPAREN LBRACE struct_lit_opt RBRACE
   ↪  { StructLit($2, $5) }*/

/*struct_lit_opt:
    nothing { [] }
  | struct_lit_list { List.rev $1 }*/

/*struct_lit_list:
    PERIOD ID ASSIGN expr { [($2, $4)] }
  | struct_lit_list COMMA PERIOD ID ASSIGN expr { ($4, $6) :: $1 }*/

actuals_opt:
    /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

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

/* rows:
    actuals_opt             { [$1] }
  | rows SEMI actuals_opt   {$3 :: $1} */

rows:
    LBRACK actuals_opt RBRACK              { [$2] }
  | rows COMMA LBRACK actuals_opt RBRACK   { $4 :: $1 }
```

ast.ml

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

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

type uop = Neg | Not | Transpose

(* TODO: to support nested structs, will want to def StructAccess of
 ↪  (StructAccess * string) *)
(* type struct_access = string * string *)

(* Types *)
type primitive_type = Float | Int | Bool | Void | String | Imatrix | Fmatrix
                      (* Tuple *)
```

```
14  type typ =
15      PrimitiveType of primitive_type
16    | StructType of string
17    | FptrType of typ list
18    | ArrayType of typ
19
20  type location = Local | External
21
22  type bind = typ * string
23
24  type expr =
25      IntLit of int
26    | FloatLit of float
27    | StringLit of string
28    | BoolLit of bool
29    | Id of string
30    | Binop of expr * op * expr
31    | Unop of uop * expr
32    | Assign of string * expr
33    | Call of string * expr list
34    | StructAccess of (string * string)
35    | StructAssign of (string * string * expr)
36    | ArrayAccess of (string * expr)   (* only allow 1-dim arrays *)
37    | ArrayAssign of (string * expr * expr)
38    | MakeStruct of typ
39    | MakeArray of (typ * expr)
40    | ArrayLit of (typ * expr list)
41    | Pipe of (expr * expr)
42    | Dispatch of (string * string * (expr list))
43    | MatLit of (expr list list)
44    | MatIndex of (string * expr * expr)
45    | MatIndexAssign of (string * expr * expr * expr)
46    | StructArrayAccess of (string * string * expr)
47    | StructArrayAssign of (string * string * expr * expr)
48    (* | StructLit of (typ * (string * expr) list) *)
49    | Null
50    | Noexpr
51
52  type stmt =
53      Block of stmt list
54    | Expr of expr
55    | Return of expr
56    | If of expr * stmt * stmt
57    | For of expr * expr * expr * stmt
58    | While of expr * stmt
59
60  type func_decl = {
```

```
    typ : typ;
    fname : string;
    formals : bind list;
    locals : bind list;
    body : stmt list;
    location : location;
}

type struct_decl = {
    name : string;
    members : bind list;
}

type fptr_type = {
  rt: typ;
  args: typ list;
}

type program = {
    global_vars: bind list;
    functions: func_decl list;
    structs: struct_decl list;
}
```

semant.ml

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

open Ast
open Util

module StringMap = Map.Make(String)

(* Semantic checking of a program. Returns void if successful,
   throws an exception if something is wrong.

   Check each global variable, then check each function *)

let check program =

  let globals = program.global_vars
  and functions = program.functions
  and structs = program.structs in

(*============================ Checking Globals ==============================*)
```

```
20    List.iter (check_not_void (fun n -> "illegal void global " ^ n)) globals;
21    (* List.iter (check_no_structs (fun n -> "illegal struct global " ^ n))
      ↪  globals; *)
22
23    (* TODO: support global structs. To do this construct struct definitions first
      ↪  in codegen *)
24    (* List.iter (check_no_opaque (fun n -> "opaque struct " ^ n)) globals; *)
25
26    report_duplicate (fun n -> "duplicate global " ^ n) (List.map snd globals);
27
28  (*============================ Checking Structs ==============================*)
29  (* TODO: struct empty fail test, struct duplicate fail test *)
30  (* TODO: passing struct info function test *)
31    List.iter (check_struct_not_empty (fun n -> "empty struct " ^ n)) structs;
32    (* List.iter (check_struct_no_nested (fun n -> "nested struct " ^ n)) structs;
      ↪  *)
33    report_duplicate (fun n -> "duplicate struct name: " ^ n)
34      (List.map (fun s -> s.name) structs);
35
36    let struct_decls = List.fold_left (fun m sd -> StringMap.add sd.name sd m)
37                         StringMap.empty structs in
38
39  (*============================ Checking Functions ============================*)
40    let built_in_keywords = Array.to_list
41      [|
42        "make"; "len"; "free"; "free_arr"; "size"; "memset"; "memcpy";
43        "concat"; "append";
44      |]
45    in
46
47    List.iter (fun fname ->
48      if List.mem fname (List.map (fun fd -> fd.fname) functions)
49      then raise (Failure ("function " ^ fname ^ " may not be defined"))
50    ) built_in_keywords;
51
52    report_duplicate (fun n -> "duplicate function " ^ n)
53      (List.map (fun fd -> fd.fname) functions);
54
55    (* Function declaration for a named function *)
56    (* let built_in_decls = StringMap.singleton "printbig"
57       { typ = Void; fname = "printbig"; formals = [(Int, "x")]; locals = []; body
  ↪   = [] }
58    in *)
59    let built_in_decls = StringMap.empty in
60
61    let function_decls = List.fold_left (fun m fd -> StringMap.add fd.fname fd m)
62                           built_in_decls functions
```

```
63   in
64
65   let function_decl s = try StringMap.find s function_decls
66        with Not_found -> raise (Failure ("unrecognized function " ^ s))
67   in
68
69   let _ = function_decl "main" in (* Ensure "main" is defined *)
70
71   let check_function func =
72
73     (* print_endline "hello";
74
75     List.iter (fun (t,s) -> print_endline (string_of_typ t ^ s)) func.formals;
↪     *)
76
77     List.iter (check_not_void (fun n -> "illegal void formal " ^ n ^
78       " in " ^ func.fname)) func.formals;
79
80     report_duplicate (fun n -> "duplicate formal " ^ n ^ " in " ^ func.fname)
81       (List.map snd func.formals);
82
83     List.iter (check_not_void (fun n -> "illegal void local " ^ n ^
84       " in " ^ func.fname)) func.locals;
85
86     report_duplicate (fun n -> "duplicate local " ^ n ^ " in " ^ func.fname)
87       (List.map snd func.locals);
88
89     (* Type of each variable (global, formal, or local *)
90     (* TODO: add support for global structs *)
91     let symbols = List.fold_left (fun m (t, n) -> StringMap.add n t m)
92             StringMap.empty (globals @ func.formals @ func.locals )
93     in
94
95     let type_of_identifier s =
96       try StringMap.find s symbols
97       with Not_found -> raise (Failure ("undeclared identifier " ^ s))
98     in
99
100    let get_struct_decl s =
101      match type_of_identifier s with
102        StructType(s_name) -> (
103          try StringMap.find s_name struct_decls
104          with Not_found -> raise (Failure ("undeclared identifier " ^ s))
105        )
106      | _ -> raise (Failure ("Not a struct " ^ s))
107    in
108
```

```
109     (* ========================================================================== *)
110
111     (* Return the type of an expression or throw an exception *)
112     let rec expr : expr -> typ = function
113               IntLit _ -> PrimitiveType(Int)
114       | FloatLit _ -> PrimitiveType(Float)
115       | BoolLit _ -> PrimitiveType(Bool)
116       | StringLit _ -> PrimitiveType(String)
117       | Noexpr -> PrimitiveType(Void)
118       | Null -> PrimitiveType(Void)
119       | Id s ->
120         let ret_typ =
121         try
122           type_of_identifier s
123         with _ ->  (*try searching for function ptr *)
124           try
125             let fdecl = function_decl s in
126               let rt_typ = fdecl.typ
127               and form_typs = List.map (fun (typ, _) -> typ) fdecl.formals in
128                 FptrType (List.append form_typs [rt_typ])
129           with _ -> raise (Failure ("undeclared identifier " ^ s))
130         in ret_typ
131       | Pipe (e1, e2) ->
132         begin
133           match e2 with
134             Call(fname, actuals) -> expr (Call(fname, e1 :: actuals))
135           | _ -> raise (Failure
136               ("cannot pipe " ^ string_of_expr e1 ^
137               " into expression" ^ string_of_expr e2))
138         end
139       | Dispatch(s_name, mthd_name, el) ->
140         let s_decl = get_struct_decl s_name in
141         let real_method = methodify mthd_name s_decl.name in
142         expr (Call(real_method, (Id(s_name)) :: el))
143         (* TEMP: TODO: Add checking *)
144       | MatLit (m) as ex ->
145         let c = List.length (List.hd m) in
146         let check l = if List.length l != c then raise (Failure ("Matrix
            ↪  literal cannot be jagged in " ^ string_of_expr ex)) in
147         List.iter check m;
148         PrimitiveType(Fmatrix)
149       | MatIndex(mat, e2, e3) as ex ->
150         let fm = (type_of_identifier mat)
151         and i = expr e2
152         and j = expr e3 in
153         if
154           match_primitive [|Fmatrix; Imatrix|] fm &&
```

```
155           match_primitive [|Int|] i &&
156           match_primitive [|Int|] j
157         then
158           if match_primitive [|Fmatrix|] fm
159           then PrimitiveType(Float)
160           else PrimitiveType(Int)
161         else
162           raise (Failure ("Illegal attempt to index matrix in expr " ^
              ↪  string_of_expr ex))
163       | MatIndexAssign(mat, e2, e3, e4) as ex ->
164         let typ = expr (MatIndex(mat, e2, e3)) in
165         check_assign typ (expr e4) ex
166       | MakeStruct (typ) as ex ->
167           if match_struct typ then typ else
168           raise (Failure  ("illegal make, must be type struct, in " ^
              ↪  string_of_expr ex))
169       | MakeArray (typ, e) as ex ->
170           if match_primitive [|Int|] (expr e) then ArrayType(typ) else
171           raise (Failure  ("illegal make, must provide integer size, in " ^
              ↪  string_of_expr ex))
172       | StructAccess (s_name, member) -> ignore(type_of_identifier s_name);
          ↪  (*check it's declared *)
173           let s_decl = get_struct_decl s_name in (* get the ast struct_decl type
              ↪  *)
174           get_struct_member_type s_decl member
175           ("Illegal struct member access: " ^ s_name  ^ "." ^ member)
176       | StructAssign (s_name, member, e) as ex ->  (* TODO: add illegal assign
          ↪  test *)
177           let t = expr e and struct_decl = get_struct_decl s_name in
178           let member_t = get_struct_member_type struct_decl member
179               ("Illegal struct member access: " ^ s_name  ^ "." ^ member) in
180           check_assign member_t t ex
181       | ArrayAccess (a_name, _) ->
182           let t = type_of_identifier a_name in
183           check_array_or_throw t a_name;
184           get_array_type t
185       | ArrayAssign (a_name, _, e) as ex ->
186           let t = (type_of_identifier a_name)
187           and expr_t = (expr e) in
188           check_array_or_throw t a_name;
189           let arr_t = get_array_type t in
190           check_assign arr_t expr_t ex
191       | ArrayLit(arr_type, expr_list) as ex ->
192         if match_array arr_type then
193           let inner_type = get_array_type arr_type in
194           List.iter (fun e -> ignore(check_assign inner_type (expr e) ex))
              ↪  expr_list;
```

```
195           arr_type
196         else raise (Failure ("expected array type in expr " ^ string_of_expr
            ↪  ex))
197       | StructArrayAccess(s_name, member, idx_expr) as ex ->
198         let t = expr (StructAccess(s_name, member))
199         and idx = expr idx_expr in
200         if match_array t && match_primitive [|Int|] idx then get_array_type t
201         else raise (Failure ("struct field is not an array in " ^
            ↪  string_of_expr ex))
202       | StructArrayAssign(s_name, member, idx_expr, e) as ex ->
203         let t = expr (StructAccess(s_name, member))
204         and idx = expr idx_expr in
205         if match_array t && match_primitive [|Int|] idx
206         then
207           let inner_t = get_array_type t in
208           check_assign inner_t (expr e) ex
209         else raise (Failure ("struct field is not an array in " ^
            ↪  string_of_expr ex))
210       | Binop(e1, op, e2) as e -> let typ1 = expr e1 and typ2 = expr e2 in
211         let ret =
212         try
213            match (typ1, typ2) with
214               (PrimitiveType(t1), PrimitiveType(t2)) -> (
215                 let inner_type =
216                      match op with
217                         Add | Sub | Mult | Div when (t1 = t2) &&
218                           (t1 = Int || t1 = Float || t1 = String || t1 =
                              ↪  Imatrix || t1 = Fmatrix) -> t1
219                       | Mod when (t1 = t2) && (t1 = Int) -> t1
220                       | Dot when (t1 = t2) && (t1 = Imatrix || t2 = Fmatrix) ->
                        ↪  t1
221                       (* Is it possible here to cast int scalar types into
                        ↪  double to prevent
222                       a compiler error later on? *)
223                       | Add | Sub | Mult | Div when (t1 = Fmatrix || t1 =
                        ↪  Imatrix) && (t2 = Float) -> t1
224                       | Add | Sub | Mult | Div when (t1 = Float) && (t2 =
                        ↪  Fmatrix || t2 = Imatrix) -> t2
225                       | Equal | Neq when (t1 = t2) ->
226                         let check_eq_typ = function
227                           | (Imatrix | Fmatrix)  -> Imatrix
228                           | (Float | Int | Bool) -> Bool
229                           | _                    -> raise Not_found
230                         in check_eq_typ t1
231
232                       | Less | Leq | Greater | Geq when (t1 = t2) && (t1 =
                        ↪  Float || t1 = Int) -> Bool
```

```
233                     | And | Or when (t1 = t2) && (t1 = Bool) -> Bool
234
235                     | _ -> raise Not_found
236                       (* TODO: Need to figure out return type of a boolean
                          ↪  matrix... is that just an Imatrix?? *)
237               in PrimitiveType(inner_type)
238             )
239           | _ -> raise (Failure "not implemented")
240       with Not_found -> raise (Failure ("Illegal binary operator " ^
241                         string_of_typ typ1 ^ " " ^ string_of_op op ^ " " ^
242                         string_of_typ typ2 ^ " in " ^ string_of_expr e))
243       in ret
244     | Unop(op, e) as ex ->
245        let typ1 = expr e in (
246          match typ1 with
247             PrimitiveType(t) -> (
248                let inner_type =
249                        match op with
250                           Neg when (t != String && t != Bool) -> t
251                   | Not when t = Bool -> t
252                   | Transpose when (t = Fmatrix || t = Imatrix) -> t
253                   | _ -> raise (Failure ("illegal unary operator " ^
                      ↪  string_of_uop op ^
254                                                          string_of_typ typ1 ^
                                                            ↪  " in " ^
                                                            ↪  string_of_expr
                                                            ↪  ex))
255                in PrimitiveType(inner_type)
256             )
257           | _ -> raise (Failure "not implemented")
258       )
259     | Assign(var, e) as ex ->
260       let lt = type_of_identifier var
261       and rt = expr e in
262       check_assign lt rt ex
263   (*============================ built in fns ================================*)
264     | Call("printf", _) -> PrimitiveType(Int)
265     | Call("time", _) -> PrimitiveType(Int)
266     | Call("float_of_int", [e]) as ex ->
267       let t = expr e in
268       if match_primitive [|Int|] t then PrimitiveType(Float)
269       else raise (Failure ("expected int, got type " ^ string_of_typ t ^ " in
                ↪  "
270                           ^ string_of_expr ex))
271     | Call("int_of_float", [e]) as ex ->
272       let t = expr e in
273       if match_primitive [|Float|] t then PrimitiveType(Int)
```

```
274         else raise (Failure ("expected float, got type " ^ string_of_typ t ^ "
             ↪  in "
275                              ^ string_of_expr ex))
276       | Call("is_null", _)  -> PrimitiveType(Bool)
277       | Call("len", [e]) ->
278         let t = expr e in
279         if match_array t then PrimitiveType(Int)
280         else raise (Failure ("Illegal argument of type " ^ string_of_typ t ^ "
             ↪  to len, must be array"))
281       | Call("free", [e]) ->
282         let t = expr e in
283         if (match_struct t || match_primitive [|Imatrix; Fmatrix|] t)
284         then PrimitiveType(Void)
285         else raise (Failure ("Illegal argument of type " ^ string_of_typ t ^ "
             ↪  to free, must be struct or matrix"))
286       | Call("free_arr", [e]) ->
287         let t = expr e in
288         if (match_array t)
289         then PrimitiveType(Void)
290         else raise (Failure ("Illegal argument of type " ^ string_of_typ t ^ "
             ↪  to free_arr, must be array"))
291       | Call("concat", [e1; e2]) as ex ->
292         let arr1 = expr e1
293         and arr2 = expr e2 in
294         if match_array arr1 && match_array arr2
295         then check_assign arr1 arr2 ex
296         else raise (Failure ("Illegal arguments of types " ^
297           string_of_typ arr1 ^ "and" ^ string_of_typ arr2 ^
298           " to concat, must be arrays"))
299       | Call("append", [e1; e2]) as ex ->
300         let arr = expr e1
301         and elem = expr e2 in
302         ignore (check_assign (get_array_type arr) elem ex); arr
303       | Call("rows", [e]) as ex ->
304         let fm = expr e in
305         if match_primitive [|Fmatrix|] fm
306         then PrimitiveType(Int)
307         else raise (Failure ("non matrix argument to rows in " ^ string_of_expr
             ↪  ex))
308       | Call("cols", [e]) as ex ->
309         let fm = expr e in
310         if match_primitive [|Fmatrix|] fm
311         then PrimitiveType(Int)
312         else raise (Failure ("non matrix argument to cols in " ^ string_of_expr
             ↪  ex))
313     (*============================================================================*)
314       | Call(fname, actuals) as call ->
```

```
315         try (* first check if it is a function pointer arg *)
316           let var = type_of_identifier fname in
317             match var with
318               FptrType(fp) ->
319                 let (args, rt) = parse_fptr_type fp in
320                   if List.length actuals != List.length args then
321                     raise (Failure ("Fail"))
322                   else
323                     List.iter2 (fun ft e -> let et = expr e in
324                       ignore (check_func_param_assign ft et (Failure
                            ↪  ("Fail"))))
325                       args actuals;
326                   rt
327             | _ -> raise (Failure ("Fail"))
328         with _ ->
329           let fd = function_decl fname in
330            if List.length actuals != List.length fd.formals then
331              raise (Failure ("expecting " ^ string_of_int
332                (List.length fd.formals) ^ " arguments in " ^ string_of_expr
                   ↪  call))
333            else
334              List.iter2 (fun (ft, _) e -> let et = expr e in
335                 ignore (check_func_param_assign ft et
336                   (Failure ("illegal actual argument found " ^ string_of_typ et
                        ↪  ^
337                   " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr
                        ↪  e))))
338                fd.formals actuals;
339              fd.typ
340     in
341
342     let check_bool_expr e = if not (match_primitive [|Bool|] (expr e))
343      then raise (Failure (
344        "expected Boolean expression in " ^ string_of_expr e
345        ))
346      else () in
347
348     (* Verify a statement or throw an exception *)
349     let rec stmt = function
350         Block sl -> let rec check_block = function
351            [Return _ as s] -> stmt s
352          | Return _ :: _ -> raise (Failure "nothing may follow a return")
353          | Block sl :: ss -> check_block (sl @ ss)
354          | s :: ss -> stmt s ; check_block ss
355          | [] -> ()
356         in check_block sl
357       | Expr e -> ignore (expr e)
```

```
      | Return e -> let t = expr e in
          if (check_asn_silent t func.typ) then () else
          raise (Failure ("return gives " ^ string_of_typ t ^ " expected " ^
                          string_of_typ func.typ ^ " in " ^ string_of_expr e))
      | If(p, b1, b2) -> check_bool_expr p; stmt b1; stmt b2
      | For(e1, e2, e3, st) -> ignore (expr e1); check_bool_expr e2;
                               ignore (expr e3); stmt st
      | While(p, s) -> check_bool_expr p; stmt s
    in

    stmt (Block func.body)

  in
  List.iter check_function functions
```

codegen.ml

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

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

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

Detailed documentation on the OCaml LLVM library:

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

*)

module L = Llvm
module A = Ast
module U = Util

module P = Printf

module StringMap = Map.Make(String)

exception Bug of string;;

let translate program =
  let globals = program.A.global_vars
  and functions = program.A.functions
  and structs = program.A.structs in
```

```
29
30    let context = L.global_context () in
31    let the_module = L.create_module context "MicroC"
32
33  (* ============================== Types ================================== *)
34    and float_t = L.double_type context
35    and i32_t  = L.i32_type  context
36    and i8_t   = L.i8_type   context
37    and i1_t   = L.i1_type   context
38    and void_t = L.void_type context in
39
40    let string_t = L.pointer_type i8_t
41    and i32_ptr_t = L.pointer_type i32_t
42    and i8_ptr_t = L.pointer_type i8_t
43    and fmatrix_t = L.pointer_type i32_t in
44
45    let ltype_of_primitive_type = function
46        A.Int -> i32_t
47      | A.Float -> float_t
48      | A.String -> string_t
49      | A.Bool -> i1_t
50      | A.Void -> void_t
51      | A.Fmatrix -> fmatrix_t
52      | A.Imatrix -> fmatrix_t
53    in
54
55    let rec ltype_of_typ struct_decl_map = function
56        A.PrimitiveType(primitive) -> ltype_of_primitive_type(primitive)
57      | A.StructType(s) -> L.pointer_type (fst (StringMap.find s
        ↪  struct_decl_map))
58      | A.ArrayType(typ) ->
59          if (U.match_struct typ || U.match_array typ)
60          then ltype_of_typ struct_decl_map typ  (* already a pointer, don't cast
            ↪  *)
61          else L.pointer_type (ltype_of_typ struct_decl_map typ)
62      | A.FptrType(fp) ->
63          let (arg_typs, ret_typ) = U.parse_fptr_type fp in
64          let rt = ltype_of_typ struct_decl_map ret_typ
65          and args = Array.of_list
66            (List.map (fun t -> ltype_of_typ struct_decl_map t) arg_typs) in
67          L.pointer_type (L.function_type rt args)
68    in
69  (* ========================================================================== *)
70    (* Collect struct declarations. Builds a map struct_name[string] -> (lltype,
      ↪  A.struct_decl)  *)
71    let struct_decl_map =
72      let add_struct m struct_decl =
```

```
73      let name = struct_decl.A.name
74      and members = Array.of_list
75        (List.map (fun (t, _) -> ltype_of_typ m t) struct_decl.A.members) in
76      let struct_type = L.named_struct_type context ("struct."^name) in
77        L.struct_set_body struct_type members false;
78      (* let struct_type = L.struct_type context members in (* TODO: use named
        ↪  or unnamed structs? *) *)
79      StringMap.add name (struct_type, struct_decl) m in
80    List.fold_left add_struct StringMap.empty structs in
81
82  let struct_lltype_list =
83    let bindings = StringMap.bindings struct_decl_map in
84    List.map (fun (_, (lltype, _)) -> L.pointer_type lltype) bindings
85  in
86
87  (* determines if the lltype is a ptr to struct type *)
88  let is_ptr_to_struct llval =
89    let lltype = L.type_of llval in
90    U.contains lltype struct_lltype_list
91  in
92
93  let get_struct_pointer_lltype llval =
94    let lltype = L.type_of llval in
95    U.try_get lltype struct_lltype_list
96  in
97
98  (* function used to initialize global and local variables *)
99  let empty_string = L.define_global "__empty_string" (L.const_stringz context
    ↪  "") the_module in
100 let init_var = function
101     A.PrimitiveType(typ) -> (
102       match typ with
103         A.Float -> L.const_float float_t 0.0
104       | A.Bool -> L.const_int i1_t 0
105       | A.String -> L.const_bitcast empty_string string_t
106       | A.Imatrix -> L.const_null fmatrix_t
107       | A.Fmatrix -> L.const_null fmatrix_t
108       (* TODO: jayz, what are the default types here? *)
109       | _ -> L.const_int i32_t 0
110     )
111   | A.StructType(_) as typ -> L.const_null (ltype_of_typ struct_decl_map typ)
112   | A.ArrayType(_) as typ -> L.const_null (ltype_of_typ struct_decl_map typ)
113   | A.FptrType(_) as typ -> L.const_null (ltype_of_typ struct_decl_map typ)
114 in
115
116 (* Declare each global variable; remember its value in a map *)
117 (* Map variable_name[string] --> (llvalue, A.typ) *)
```

```
118   let global_vars =
119     let global_var m (t, name) =
120       let init = init_var t in
121       let global_llvalue = L.define_global name init the_module in
122       StringMap.add name (global_llvalue, t) m in
123     List.fold_left global_var StringMap.empty globals in
124
125 (*====================== EXTERNAL FUNCTION DECLARATIONS ====================*)
126   let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
127   let printf_func = L.declare_function "printf" printf_t the_module in
128
129   let time_t = L.var_arg_function_type i32_t [| L.pointer_type i32_t |] in
130   let time_func = L.declare_function "time" time_t the_module in
131
132   (* let memset_t = L.function_type void_t [| i8_ptr_t; i32_t; i32_t|] in
133   let memset = L.declare_function "memset" memset_t the_module in *)
134
135   let memcpy_t = L.function_type i32_t [| i8_ptr_t; i8_ptr_t; i32_t|] in
136   let memcpy = L.declare_function "memcpy" memcpy_t the_module in
137 (*==========================================================================*)
138   (* TODO: jz matliteral *)
139   let init_fmat_literal_t = L.var_arg_function_type fmatrix_t [| L.pointer_type
      ↪  float_t; i32_t; i32_t; |] in
140   let init_fmat_literal_func = L.declare_function "init_fmat_literal"
      ↪  init_fmat_literal_t the_module in
141
142   (*
143   Define each function (arguments and return type) so we can call it
144   Builds a map fname [string] -> ([llvalue], [Ast.func_decl])
145   *)
146   let local_functions =
147     List.filter (fun fdecl -> fdecl.A.location = A.Local) functions
148   and extern_functions =
149     List.filter (fun fdecl -> fdecl.A.location = A.External) functions in
150
151   let extern_decls =
152     let extern_decl m fdecl =
153       let name = fdecl.A.fname and
154         formal_types = Array.of_list
155           (List.map(fun (t, _) -> ltype_of_typ struct_decl_map t)
            ↪  fdecl.A.formals) in
156       let ftype = L.function_type (ltype_of_typ struct_decl_map fdecl.A.typ)
        ↪  formal_types in
157       StringMap.add name (L.declare_function name ftype the_module, fdecl) m in
158     List.fold_left extern_decl StringMap.empty extern_functions in
159
160   let local_decls =
```

```
161     let function_decl m fdecl =
162       let name = fdecl.A.fname
163       and formal_types = Array.of_list
164         (List.map (fun (t,_) -> ltype_of_typ struct_decl_map t)
          ↪  fdecl.A.formals) in
165       let ftype = L.function_type (ltype_of_typ struct_decl_map fdecl.A.typ)
          ↪  formal_types in
166       StringMap.add name (L.define_function name ftype the_module, fdecl) m in
167     List.fold_left function_decl StringMap.empty local_functions in
168
169   let find_func fname =
170     if (StringMap.mem fname local_decls) then StringMap.find fname local_decls
171     else StringMap.find fname extern_decls
172   in
173
174   (* call an externally defined function by name and arguments *)
175   let build_external fname actuals the_builder =
176     let (fdef, fdecl) = (try StringMap.find fname extern_decls with
177       Not_found -> raise(Failure("Not defined: " ^ fname ))) in
178     let result = (match fdecl.A.typ with
179         A.PrimitiveType(t) when t = A.Void -> ""
180       | _ -> fname ^ "_res")
181     in
182     L.build_call fdef actuals result the_builder in
183
184   (* Fill in the body of the given function *)
185   (* TODO: need to make all structs default on heap. If initialized locally, put
      ↪  on heap.
186        If seen in function signature, treat as struct pointer
187    *)
188   let build_function_body fdecl =
189     let (the_function, _) = try StringMap.find fdecl.A.fname local_decls with
       ↪  Not_found -> raise (Bug "2") in
190     (* return an instruction builder positioned at end of formal store/loads *)
191     let builder = L.builder_at_end context (L.entry_block the_function) in
192
193     (* Construct the function's "locals": formal arguments and locally
194        declared variables.  Allocate each on the stack, initialize their
195        value, if appropriate, and remember their values in the "locals" map *)
196     let local_vars =
197       (* need to alloc and store params *)
198       let add_formal m (t, n) p =
199         L.set_value_name n p;
200               let local = L.build_alloca (ltype_of_typ struct_decl_map t) n
                  ↪  builder in
201         ignore (L.build_store p local builder);  (* local is stack pointer *)
202         StringMap.add n (local, t) m in
```

```
203       (* only need to alloc local vars *)
204       let add_local m (t, n) =
205         let local_var = L.build_alloca (ltype_of_typ struct_decl_map t) n
              ↪  builder in
206         ignore (L.build_store (init_var t) local_var builder);
207         StringMap.add n (local_var, t) m
208       in
209
210       let formals = List.fold_left2 add_formal StringMap.empty fdecl.A.formals
211           (Array.to_list (L.params the_function)) in
212     (* produces a map name[string] --> llvalue *)
213     List.fold_left add_local formals fdecl.A.locals in
214
215     (* Return the llvalue for a variable or formal argument *)
216     let lookup_llval n = try fst (StringMap.find n local_vars)
217                  with Not_found -> fst (StringMap.find n global_vars)
218     in
219
220     (* returns the A.typ for a var *)
221     let lookup_typ n = try snd (StringMap.find n local_vars)
222                  with Not_found -> snd (StringMap.find n global_vars)
223     in
224
225     (* TODO: fail test trying to access a member of an undeclared struct *)
226     let get_struct_decl s_name =
227       try
228         let typ = lookup_typ s_name in
229         match typ with
230           A.StructType(s) -> snd (StringMap.find s struct_decl_map)
231         | _ -> raise Not_found
232       with Not_found -> raise (Failure (s_name ^ " not declared"))
233     in
234
235     (* ======================== Binary Operators ======================== *)
236
237     let int_ops = function
238       A.Add     -> L.build_add
239     | A.Sub     -> L.build_sub
240     | A.Mult    -> L.build_mul
241     | A.Div     -> L.build_sdiv
242     | A.And     -> L.build_and
243     | A.Or      -> L.build_or
244     | A.Equal   -> L.build_icmp L.Icmp.Eq
245     | A.Neq     -> L.build_icmp L.Icmp.Ne
246     | A.Less    -> L.build_icmp L.Icmp.Slt
247     | A.Leq     -> L.build_icmp L.Icmp.Sle
248     | A.Greater -> L.build_icmp L.Icmp.Sgt
```

```
249     | A.Geq     -> L.build_icmp L.Icmp.Sge
250     | _         -> raise Not_found
251     in
252
253     let float_ops = function
254       A.Add     -> L.build_fadd
255     | A.Sub     -> L.build_fsub
256     | A.Mult    -> L.build_fmul
257     | A.Div     -> L.build_fdiv
258     | A.Equal   -> L.build_fcmp L.Fcmp.Ueq
259     | A.Neq     -> L.build_fcmp L.Fcmp.Une
260     | A.Less    -> L.build_fcmp L.Fcmp.Ult
261     | A.Leq     -> L.build_fcmp L.Fcmp.Ule
262     | A.Greater -> L.build_fcmp L.Fcmp.Ugt
263     | A.Geq     -> L.build_fcmp L.Fcmp.Uge
264     | _         -> raise Not_found
265     in
266
267     let bool_ops = function
268       A.And     -> L.build_and
269     | A.Or      -> L.build_or
270     | _         -> raise Not_found
271     in
272
273     let matrix_matrix_ops = function
274       A.Add     ->  "mm_add"
275     | A.Sub     ->  "mm_sub"
276     | A.Mult    ->  "mm_mult"
277     | A.Div     ->  "mm_div"
278     | A.Dot     ->  "dot"
279     | _         -> raise Not_found
280     in
281
282     let scalar_matrix_ops = function
283       A.Add     ->  "sm_add"
284     | A.Sub     ->  "sm_sub"
285     | A.Mult    ->  "sm_mult"
286     | A.Div     ->  "sm_div"
287     | A.Equal   ->  "sm_eq"
288     | A.Neq     ->  "sm_neq"
289     | A.Less    ->  "sm_lt"
290     | A.Leq     ->  "sm_leq"
291     | A.Greater ->  "sm_gt"
292     | A.Geq     ->  "sm_geq"
293     | _         -> raise Not_found
294     in
295
```

```
296
297
298     (* ========================================================================= *)
299
300     (* ========================= Array Constructors ============================ *)
301     (*
302       Whenever an array is made, we malloc and additional 16 bytes of metadata,
303       which contains size and length information. This allows us to implement
304       len() in a static context, and opens several possibilities including
305       array concatenation, dynamic array resizing, etc.
306       The layout will be:
307       +--------------+--------------+----------+--------+
308       | element size | size (bytes) | len[int] | elem1  | ...
309       +--------------+--------------+----------+--------+
310     *)
311
312     let elem_size_offset = L.const_int i32_t (-3)
313     and size_offset = L.const_int i32_t (-2)
314     and len_offset = L.const_int i32_t (-1)
315     and metadata_sz = L.const_int i32_t 12 in  (* 12 bytes overhead *)
316
317     let put_meta body_ptr offset llval builder =
318       let ptr = L.build_bitcast body_ptr i32_ptr_t "i32_ptr_t" builder in
319       let meta_ptr = L.build_gep ptr [| offset |] "meta_ptr" builder in
320       L.build_store llval meta_ptr builder
321     in
322
323     let get_meta body_ptr offset builder =
324       let ptr = L.build_bitcast body_ptr i32_ptr_t "i32_ptr_t" builder in
325       let meta_ptr = L.build_gep ptr [| offset |] "meta_ptr" builder in
326       L.build_load meta_ptr "meta_data" builder
327     in
328
329     let meta_to_body meta_ptr builder =
330       let ptr = L.build_bitcast meta_ptr i8_ptr_t "meta_ptr" builder in
331       L.build_gep ptr [| (L.const_int i8_t (12)) |] "body_ptr" builder
332     in
333
334     let body_to_meta body_ptr builder =
335       let ptr = L.build_bitcast body_ptr i8_ptr_t "body_ptr" builder in
336       L.build_gep ptr [| (L.const_int i8_t (-12)) |] "meta_ptr" builder
337     in
338
339     let make_array element_t len builder =
340       let element_sz = L.build_bitcast (L.size_of element_t) i32_t "b" builder
        ↪  in
341       let body_sz = L.build_mul element_sz len "body_sz" builder in
```

```
342       let malloc_sz = L.build_add body_sz metadata_sz "make_array_sz" builder
          ↪  in
343       let meta_ptr = L.build_array_malloc i8_t malloc_sz "make_array" builder
          ↪  in
344       let body_ptr = meta_to_body meta_ptr builder in
345       ignore (put_meta body_ptr elem_size_offset element_sz builder);
346       ignore (put_meta body_ptr size_offset malloc_sz builder);
347       ignore (put_meta body_ptr len_offset len builder);
348       L.build_bitcast body_ptr (L.pointer_type element_t) "make_array_ptr"
          ↪  builder
349     in
350
351     (* TODO: free old array before append, and handle edge case where we are
352     appending a pointer to struct within the existing array.
353     *)
354     let arr_copy_and_free arr_gep elem_ptr elem_sz restore_ptr builder =
355       let casted_elem_ptr = L.build_bitcast elem_ptr i8_ptr_t
          ↪  "struct_to_char_ptr" builder
356       and casted_arr_ptr = L.build_bitcast arr_gep i8_ptr_t "arr_to_char_ptr"
          ↪  builder
357       and casted_elem_sz = L.build_bitcast elem_sz i32_t "i64_to_i32" builder
          ↪  in
358       ignore(L.build_call memcpy [|casted_arr_ptr; casted_elem_ptr;
          ↪  casted_elem_sz|] "" builder);
359       ignore(L.build_free casted_elem_ptr builder); (* free original object *)
360       match restore_ptr with
361         Some ptr -> L.build_store arr_gep ptr builder
362       | None -> elem_ptr
363     in
364
365     (* allocates an additional elem_sz bytes to array and memcpys
366       Note: appending a struct will
367       1. memcpy original struct to array
368       2. free original struct
369       3. have struct ptr now point to the element in the array.
370
371       Note: it is illegal to have a pointer to a struct within an arr, and then
372       append that internal struct to the end of the array. This breaks because
373       it causes a double free: once to free the original array, and another
374       to free the strict. It is VERY difficult to track statically whether or
375       not a pointer is pointing within array bounds, so we will just not support
376       this.
377
378       @param restore_ptr: the stack address of the pointer to an element we are
379       appending. Optional. Only populated if we are appending a struct. This is
380       so we may reset the struct ptr to the newly malloced array element.
381     *)
```

```
382     let array_append arr_ptr elem_val restore_ptr builder =
383       let orig_sz = get_meta arr_ptr size_offset builder
384       and elem_sz = get_meta arr_ptr elem_size_offset builder
385       and orig_len = get_meta arr_ptr len_offset builder
386       and src_meta_ptr = body_to_meta arr_ptr builder in
387       let new_len = L.build_add orig_len (L.const_int i32_t 1)
          ↪  "post_append_len" builder in
388       let new_sz = L.build_add orig_sz elem_sz "post_append_sz" builder in
389       let dst_meta_ptr = L.build_array_malloc i8_t new_sz "append_array"
          ↪  builder in
390       let dst_body_ptr = meta_to_body dst_meta_ptr builder in
391       (* memcpy and update metadata, then free old array *)
392       ignore(L.build_call memcpy [|dst_meta_ptr; src_meta_ptr; orig_sz; |] ""
          ↪  builder);
393       ignore(put_meta dst_body_ptr size_offset new_sz builder);
394       ignore(put_meta dst_body_ptr len_offset new_len builder);
395       ignore(L.build_free src_meta_ptr builder);
396       (* now we need to copy the elem_val into the buffer *)
397       let ret_ptr = L.build_bitcast dst_body_ptr (L.type_of arr_ptr)
          ↪  "append_array_ptr" builder in
398       let arr_gep = L.build_in_bounds_gep ret_ptr [|orig_len|] "append_idx"
          ↪  builder in
399       ignore (
400       if is_ptr_to_struct elem_val
401       then (* cannot just store; we need to do a memcpy *)
402         arr_copy_and_free arr_gep elem_val elem_sz restore_ptr builder
403       else
404         L.build_store elem_val arr_gep builder
405       );
406       ret_ptr
407     in
408
409     (*
410     During concat(a, b) we will malloc a new array of len(a) + len(b) and
411     memcpy the contents of a and b.
412     We then free the original array a, and leave b untouched.
413     usage: a = concat(a, b)
414     *)
415     let array_concat left_arr_ptr right_arr_ptr builder =
416       let left_meta_ptr = body_to_meta left_arr_ptr builder
417       and right_casted_ptr = L.build_bitcast right_arr_ptr i8_ptr_t "" builder
418       and left_arr_sz = get_meta left_arr_ptr size_offset builder
419       and right_arr_sz =
420         L.build_sub
421           (get_meta right_arr_ptr size_offset builder)
422           metadata_sz "minus_meta_sz" builder
423       and left_arr_len = get_meta left_arr_ptr len_offset builder
```

```
424       and right_arr_len = get_meta right_arr_ptr len_offset builder in
425       let new_sz = L.build_add left_arr_sz right_arr_sz "concat_sz" builder in
426       let new_len = L.build_add left_arr_len right_arr_len "concat_len" builder
           ↪  in
427       let dst_meta_ptr = L.build_array_malloc i8_t new_sz "concat_array"
           ↪  builder in
428       let dst_body_ptr = meta_to_body dst_meta_ptr builder in
429       let ret_ptr = L.build_bitcast dst_body_ptr (L.type_of left_arr_ptr)
           ↪  "concat_ret_ptr" builder in
430       let dst_concat_ptr =
431         L.build_bitcast
432           ( L.build_in_bounds_gep ret_ptr [|left_arr_len|] "" builder )
433           i8_ptr_t
434           "concat_pos_ptr"
435           builder
436       in
437       ignore(L.build_call memcpy [|dst_meta_ptr; left_meta_ptr; left_arr_sz|]
           ↪  "" builder);
438       ignore(L.build_call memcpy [|dst_concat_ptr; right_casted_ptr;
           ↪  right_arr_sz|] "" builder);
439       ignore(put_meta dst_body_ptr size_offset new_sz builder);
440       ignore(put_meta dst_body_ptr len_offset new_len builder);
441       ignore(L.build_free left_meta_ptr builder);
442       ret_ptr
443     in
444
445
446     (* ============================================================== *)
447
448     (* Construct code for an expression; return its value *)
449     let rec expr builder = function
450         A.IntLit i            -> L.const_int i32_t i
451       | A.FloatLit f          -> L.const_float float_t f
452       | A.StringLit s         -> L.build_global_stringptr (Scanf.unescaped s)
           ↪  "str" builder
453       | A.BoolLit b           -> L.const_int i1_t (if b then 1 else 0)
454       | A.Noexpr              -> L.const_int i32_t 0
455       | A.Null                -> L.const_pointer_null void_t
456       | A.Id s                ->
457         let ret =
458         try
459           L.build_load (lookup_llval s) s builder
460         with Not_found -> (* then it's probably a function pointer *)
461           fst (find_func s)
462         in ret
463       | A.Pipe(e1, e2) ->
464         begin
```

```
465           match e2 with
466             A.Call(fname, actuals) -> expr builder (A.Call(fname, e1 ::
                ↪  actuals))
467           | _ -> raise (Failure "illegal pipe")  (* this should never execute
                ↪  *)
468         end
469       | A.Dispatch(s_name, mthd_name, el) ->
470         let struct_decl = get_struct_decl s_name in
471         let real_method = U.methodify mthd_name struct_decl.A.name in
472         expr builder (A.Call(real_method, (A.Id(s_name)) :: el))
473       | A.MatIndex(mat, e2, e3) ->
474         let fm = expr builder (A.Id(mat))
475         and i = expr builder e2
476         and j = expr builder e3 in
477         build_external "mat_index" [|fm; i; j|] builder
478       | A.MatIndexAssign(mat, e2, e3, e4) ->
479         let fm = expr builder (A.Id(mat))
480         and i = expr builder e2
481         and j = expr builder e3
482         and f = expr builder e4 in
483         build_external "mat_index_assign" [|fm; i; j; f|] builder
484       | A.MakeArray(typ, e) ->
485         let len = expr builder e
486         and element_t =
487           if U.match_struct typ || U.match_array typ
488           then L.element_type (ltype_of_typ struct_decl_map typ)
489           else ltype_of_typ struct_decl_map typ
490         in
491         make_array element_t len builder
492       | A.MakeStruct(typ) ->
493         let llname = "make_struct"
494         and struct_t = L.element_type (ltype_of_typ struct_decl_map typ) in
495         L.build_malloc struct_t llname builder
496       | A.ArrayAccess (arr_name, idx_expr) ->
497         let idx = expr builder idx_expr in
498         let llname = arr_name ^ "[" ^ L.string_of_llvalue idx ^ "]" in
499         let arr_ptr_load =
500           if U.is_struct_access arr_name
501             then  (* this is to handle foo.arr[1]  TODO: currently
                      ↪  nonfunctional. parse ambiguity *)
502               let (s_name, member) = U.parse_struct_access arr_name in
503               expr builder (A.StructAccess(s_name, member))
504             else (* this is to handle normal arr[1] *)
505               let arr_ptr = lookup_llval arr_name in
506               L.build_load arr_ptr arr_name builder in
507         let arr_gep = L.build_in_bounds_gep arr_ptr_load [|idx|] llname builder
            ↪  in
```

```
508         let arr_typ = U.get_array_type (lookup_typ arr_name) in
509         (* If it's a pointer type, i.e. struct/array don't load *)
510         if U.match_struct arr_typ || U.match_array arr_typ then arr_gep
511         else L.build_load arr_gep (llname ^ "_load") builder
512       | A.ArrayAssign (arr_name, idx_expr, val_expr) ->
513         let idx = (expr builder idx_expr)
514         and assign_val = (expr builder val_expr) in
515         let llname = arr_name ^ "[" ^ L.string_of_llvalue idx ^ "]" in
516         let arr_ptr = lookup_llval arr_name in
517         let arr_ptr_load = L.build_load arr_ptr arr_name builder in
518         let arr_gep = L.build_in_bounds_gep arr_ptr_load [|idx|] llname builder
            ↪  in
519         (
520         match get_struct_pointer_lltype assign_val with
521           (* Note:
522             assigning a struct will memcpy the contents of the struct over,
523             free the former struct, and have the struct ptr now point to the
524             array element.
525             We do this to prevent assignment "implicitly" duplicating memory
526           *)
527           Some struct_ptr ->
528             let elem_type = L.element_type struct_ptr in
529             let elem_sz = L.size_of elem_type in
530             let restore_ptr =
531             (
532               match (U.try_get_id_str val_expr) with
533                 Some s -> Some (lookup_llval s)
534                 (*TODO: match case for struct literals *)
535               | None -> None
536             ) in
537             arr_copy_and_free arr_gep assign_val elem_sz restore_ptr builder
538         | None -> L.build_store assign_val arr_gep builder
539         )
540       | A.StructAccess (s_name, member) ->
541         let struct_ptr = lookup_llval s_name
542         and llname = (s_name ^ "." ^ member)
543         and struct_decl = get_struct_decl s_name in
544         let struct_ptr_load = L.build_load struct_ptr ("struct."^s_name)
            ↪  builder in
545         let struct_gep =
546           L.build_struct_gep struct_ptr_load (U.get_struct_member_idx
              ↪  struct_decl member)
547           llname builder in
548         L.build_load struct_gep (llname ^ "_load") builder
549       | A.StructAssign (s_name, member, e) ->
550           let e' = expr builder e
551           and struct_ptr = lookup_llval s_name
```

```
552           and llname = (s_name ^ "." ^ member)
553           and struct_decl = get_struct_decl s_name in
554           let struct_ptr_load = L.build_load struct_ptr ("struct."^s_name)
               ↪  builder in
555           let struct_gep =
556             L.build_struct_gep struct_ptr_load (U.get_struct_member_idx
               ↪  struct_decl member)
557             llname builder in
558         ignore (L.build_store e' struct_gep builder); e'
559       | A.ArrayLit (arr_type, expr_list) ->
560         let arr_ptr = expr builder (A.MakeArray(U.get_array_type arr_type,
561                                   A.IntLit(List.length expr_list))) in
562         List.iteri (fun idx e ->  (* TODO: need to make this work with struct
               ↪  literals *)
563           let arr_gep = L.build_in_bounds_gep arr_ptr [| L.const_int i32_t
               ↪  (idx) |] "array_lit" builder in
564           let assign_val = expr builder e in
565           ignore (L.build_store assign_val arr_gep builder)
566         ) expr_list;
567         arr_ptr
568       | A.MatLit m ->
569         let r = expr builder (A.IntLit(List.length m))
570         and c = expr builder (A.IntLit(List.length (List.hd m)))
571         and a = expr builder (A.ArrayLit(A.ArrayType(A.PrimitiveType(A.Float)),
               ↪  List.concat m)) in
572         (L.build_call init_fmat_literal_func [|a; r; c;|] "m_lit" builder)
573       | A.StructArrayAccess(s_name, member, e) ->
574           let struct_member = expr builder (A.StructAccess(s_name, member))
575           and idx = expr builder e
576           and llname = s_name ^ "." ^ member ^ "[]"
577           and struct_decl = get_struct_decl s_name in
578           let arr_gep = L.build_in_bounds_gep struct_member [|idx|] llname
               ↪  builder in
579           let arr_typ = U.get_struct_member_type struct_decl member "not
               ↪  found" in
580           let arr_inner_typ = U.get_array_type arr_typ in
581           (* If it's a pointer type, i.e. struct/array don't load *)
582           if U.match_struct arr_inner_typ || U.match_array arr_inner_typ then
               ↪  arr_gep
583           else L.build_load arr_gep (llname ^ "_load") builder
584         | A.StructArrayAssign(s_name, member, e1, e2) ->
585           let struct_member = expr builder (A.StructAccess(s_name, member))
586           and idx = expr builder e1
587           and assign_val = expr builder e2
588           and llname = s_name ^ "." ^ member ^ "[]"
589           (* and struct_decl = get_struct_decl s_name  *)
590           in
```

```
591             let arr_gep = L.build_in_bounds_gep struct_member [|idx|] llname
                 ↪  builder in
592             (
593               match get_struct_pointer_lltype assign_val with
594                 Some struct_ptr ->
595                  let elem_type = L.element_type struct_ptr in
596                  let elem_sz = L.size_of elem_type in
597                  let restore_ptr =
598                  (
599                    match (U.try_get_id_str e2) with
600                      Some s -> Some (lookup_llval s)
601                      (*TODO: match case for struct literals *)
602                    | None -> None
603                  ) in
604                  arr_copy_and_free arr_gep assign_val elem_sz restore_ptr
                      ↪  builder
605               | None -> L.build_store assign_val arr_gep builder
606           )
607
608       | A.Binop (e1, op, e2)  ->
609           let e1' = expr builder e1 and e2' = expr builder e2 in
610           let l_typ1 = L.type_of e1' and l_typ2 = L.type_of e2' in
611           let l_typs = (l_typ1, l_typ2) in
612           (
613             if      l_typs = (fmatrix_t, fmatrix_t) then (build_external
                     ↪  (matrix_matrix_ops op) [| e1'; e2'|] builder)
614             else if l_typs = (float_t, float_t) then (float_ops op e1' e2' "tmp"
                     ↪  builder)
615             else if l_typs = (i32_t, i32_t) && op = A.Mod then (build_external
                     ↪  "modulo" [|e1'; e2'|] builder)
616             else if l_typs = (i32_t, i32_t) then (int_ops op e1' e2' "tmp"
                     ↪  builder)
617             else if l_typs = (i1_t, i1_t) then (bool_ops op e1' e2' "tmp"
                     ↪  builder)
618             else if l_typ1 = fmatrix_t && (l_typ2 = i32_t || l_typ2 = float_t)
                     ↪  then
619                             let rev = expr builder (A.IntLit(1)) in
620                             (build_external (scalar_matrix_ops op) [|e1'; e2';
                                 ↪  rev|] builder)
621             else if l_typ2 = fmatrix_t && (l_typ1 = i32_t || l_typ1 = float_t)
                     ↪  then
622                             let rev = expr builder (A.IntLit(0)) in
623                             (build_external (scalar_matrix_ops op) [|e2'; e1';
                                 ↪  rev|] builder)
624             else raise (Failure ((U.string_of_op op) ^ " not defined for " ^
625                                  (L.string_of_lltype l_typ1) ^ " and " ^
626                                  (L.string_of_lltype l_typ2) ^ " in " ^
```

```
627                                 (U.string_of_expr e2)
628                               )
629                           )
630         )
631       | A.Unop(op, e) ->
632         let e' = expr builder e in
633         let l_typ = L.type_of e' in
634         (match op with
635            A.Neg          ->
636             if      l_typ = float_t then L.build_fneg e' "tmp" builder
637             else if l_typ = fmatrix_t then build_external "negate" [| e' |]
                 ↪  builder
638             else    L.build_neg e' "tmp" builder
639         | A.Not          -> L.build_not e' "tmp" builder
640         | A.Transpose    -> build_external "transpose" [| e' |] builder
641         )
642       | A.Assign (s, e) -> (* TODO: matrix reassign *)
643           let e' = expr builder e in
644           ignore (L.build_store e' (lookup_llval s) builder); e'
645     (*============================ built in fns ============================*)
646       | A.Call ("printf", act) ->
647         let actuals = List.map (expr builder) act in
648               L.build_call
649            printf_func
650            (Array.of_list actuals)
651            "printf"
652            builder
653       | A.Call("time", []) ->
654          L.build_call
655             time_func
656             [| L.const_pointer_null (L.pointer_type i32_t) |]
657             "time"
658             builder
659       | A.Call("float_of_int", [e]) ->
660          L.build_sitofp (expr builder e) float_t "float_of_int" builder
661       | A.Call("int_of_float", [e]) ->
662          L.build_fptosi (expr builder e) i32_t "int_of_float" builder
663       | A.Call("len", [e]) ->
664         let arr_ptr = expr builder e in
665         let is_null = L.build_is_null arr_ptr "null" builder in
666         L.build_select
667           is_null
668           (L.const_int i32_t 0)
669           (get_meta arr_ptr len_offset builder)
670           "len"
671           builder
672       | A.Call("free", [e]) ->
```

```
673         let ptr = expr builder e in
674         let lltype = L.type_of ptr in
675         if lltype = fmatrix_t
676         then build_external  "del_mat" [|ptr|] builder
677         else L.build_free ptr builder
678       | A.Call("free_arr", [e]) ->
679       (* we have to make a separate free for arrays to know to move ptr back
680       8 bytes so we can free metadata *)
681         let body_ptr = expr builder e in
682         let meta_ptr = body_to_meta body_ptr builder in
683         L.build_free meta_ptr builder
684       | A.Call("append", [e1; e2]) ->
685         let arr_ptr = expr builder e1
686         and elem = expr builder e2 in
687         let restore_ptr = (
688           match U.try_get_id_str e2 with
689             Some s -> Some (lookup_llval s)
690           | None -> None
691         ) in
692         array_append arr_ptr elem restore_ptr builder
693       | A.Call("concat", [e1; e2]) ->
694         let left_arr_ptr = expr builder e1
695         and right_arr_ptr = expr builder e2 in
696         array_concat left_arr_ptr right_arr_ptr builder
697     (*============================================================================*)
698       | A.Call (f_name, act) ->
699         let actuals = Array.of_list (List.rev (List.map (expr builder)
          ↪  (List.rev act))) in
700           try
701             let fdef = lookup_llval f_name in  (* first search for function
              ↪  pointer *)
702               let fptr_load = L.build_load fdef "load_fptr" builder in
703                 let result_name =
704                   if (L.return_type(L.element_type (L.element_type(L.type_of
                    ↪  fdef)))) = void_t
705                   then ""
706                   else f_name ^ "_result"
707                 in
708                 L.build_call fptr_load actuals result_name builder
709           with Not_found ->
710             (* we double reverse here for historic reasons. should we undo?
711               Need to specify the order we eval fn arguments in LRM
712             *)
713             let (fdef, fdecl) = find_func f_name in (* searching for normal
              ↪  function call *)
714               L.build_call fdef actuals (U.get_result_name f_name fdecl.A.typ)
                ↪  builder
```

```
715     in
716
717     (* Invoke "f builder" if the current block doesn't already
718        have a terminal (e.g., a branch). *)
719     let add_terminal builder f =
720       match L.block_terminator (L.insertion_block builder) with
721         Some _ -> ()
722       | None -> ignore (f builder)
723
724       in
725
726     (* Build the code for the given statement; return the builder for
727        the statement's successor *)
728     let rec stmt builder = function
729         A.Block sl -> List.fold_left stmt builder sl
730       | A.Expr e -> ignore (expr builder e); builder
731       | A.Return e -> ignore
732           (match fdecl.A.typ with
733                   A.PrimitiveType(t) when t = A.Void -> L.build_ret_void
                     ↪  builder
734             | _ -> L.build_ret (expr builder e) builder
735           ); builder
736       | A.If (predicate, then_stmt, else_stmt) ->
737           let bool_val = expr builder predicate in
738           let merge_bb = L.append_block context "merge" the_function in
739           let then_bb = L.append_block context "then" the_function in
740           add_terminal (stmt (L.builder_at_end context then_bb) then_stmt)
741                        (L.build_br merge_bb);
742           let else_bb = L.append_block context "else" the_function in
743           add_terminal (stmt (L.builder_at_end context else_bb) else_stmt)
744                        (L.build_br merge_bb);
745           ignore (L.build_cond_br bool_val then_bb else_bb builder);
746           L.builder_at_end context merge_bb
747       | A.While (predicate, body) ->
748             let pred_bb = L.append_block context "while" the_function in
749                     ignore (L.build_br pred_bb builder);
750         let body_bb = L.append_block context "while_body" the_function in
751         add_terminal (stmt (L.builder_at_end context body_bb) body)
752                           (L.build_br pred_bb);
753         let pred_builder = L.builder_at_end context pred_bb in
754         let bool_val = expr pred_builder predicate in
755               let merge_bb = L.append_block context "merge" the_function in
756               ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
757               L.builder_at_end context merge_bb
758       | A.For (e1, e2, e3, body) -> stmt builder
759           ( A.Block [A.Expr e1 ; A.While (e2, A.Block [body ; A.Expr e3]) ] )
760     in
```

```
761
762     (* Build the code for each statement in the function *)
763     let builder = stmt builder (A.Block fdecl.A.body) in
764
765     (* Add a return if the last block falls off the end *)
766     add_terminal builder (match fdecl.A.typ with
767       (* TODO: catch void *)
768         A.PrimitiveType(t) when t = A.Void -> L.build_ret_void
769       | A.PrimitiveType(t) when t = A.Float -> L.build_ret (L.const_float
          ↪  float_t 0.0)
770       | t -> L.build_ret (L.const_int (ltype_of_typ struct_decl_map t) 0)
771     )
772   in
773
774   List.iter build_function_body local_functions;
775   the_module
```

util.ml

```
1  open Ast
2
3  (*============================Pretty-printing functions==================== *)
4  let string_of_primitive_type = function
5        Int -> "int"
6      | Float -> "float"
7      | Bool -> "bool"
8      | Void -> "void"
9      | String -> "string"
10     | Imatrix -> "imatrix"
11     | Fmatrix -> "fmatrix"
12
13 let rec string_of_typ = function
14     PrimitiveType(t) -> string_of_primitive_type t
15   | StructType(s)    -> "struct " ^ s
16   | ArrayType(typ) -> (string_of_typ typ) ^ "[]"
17   | FptrType(typs) -> String.concat ", " (List.map string_of_typ typs)
18
19 let string_of_op = function
20     Add -> "+"
21   | Sub -> "-"
22   | Mult -> "*"
23   | Div -> "/"
24   | Pow -> "**"
25   | Mod -> "%"
26   | Equal -> "=="
```

```
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"
  | Dot -> ".."

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

let rec string_of_expr = function
    IntLit(l) -> string_of_int l
  | FloatLit(f) -> string_of_float f
  | StringLit(s) -> s
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | Id(s) -> s
  | Null -> "NULL"
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  (*
  | Slice(b, s, e) ->
      string_of_expr b ^ ":" ^ string_of_expr s ^ ":" ^ string_of_expr e
  | Tupselect(v, e) -> string_of_expr v ^ "[" ^ string_of_expr e ^ "]"
  | Tupassign(v, e, x) ->
      string_of_expr v ^ "[" ^ string_of_expr e ^ "] = " ^ string_of_expr x *)
  (* | Matselect(v, e1, e2) ->
      string_of_expr v ^ "[" ^ string_of_expr e1 ^ ", " ^ string_of_expr e2 ^
↪   "]"
  | Matassign(v, e1, e2, x) -> string_of_expr v ^ "[" ^ string_of_expr e1 ^
      ", " ^ string_of_expr e2 ^ "] = " ^ string_of_expr x
  | TupLit(el) -> "[" ^ String.concat ", " (List.map string_of_expr el) ^ "]"*)
  | MatLit(e1) -> "[" ^ String.concat ", " (List.map (fun e ->
       ("[" ^ String.concat ", " (List.map string_of_expr e) ^ "])) e1) ^"]"
  (* | MatLit(el) -> "[" ^ String.concat ", " (List.map (fun e -> *)
      (* "[" ^ (String.concat ", " (List.map string_of_expr e)) ^ "]" el)  ) ^
      ↪   "]" *)
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""
  | StructAccess(s_name, member) -> s_name ^ "." ^ member
```

```
72    | StructAssign(s_name, member, e) -> s_name ^ "." ^ member ^ " = " ^
      ↪  (string_of_expr e)
73    | ArrayAccess(arr_name, e) -> arr_name ^ "[" ^ string_of_expr e ^ "]"
74    | ArrayAssign(arr_name, e1, e2) -> arr_name ^ "[" ^ string_of_expr e1 ^ "]" ^
      ↪  "=" ^ string_of_expr e2
75    | MakeStruct(t) -> "make(" ^ string_of_typ t ^ ")"
76    | MakeArray(t,e) -> "make(" ^ string_of_typ t ^ "," ^ string_of_expr e ^ ")"
77    | ArrayLit(typ, el) -> "(" ^ string_of_typ typ ^ ") {" ^ String.concat ", "
      ↪  (List.map string_of_expr el) ^ "}"
78    | Pipe(e1, e2) -> string_of_expr e1 ^ " => " ^ string_of_expr e2
79    | Dispatch(strct, mthd_name, el) ->
80      strct ^ "." ^ mthd_name ^ ".(" ^ String.concat ", " (List.map
       ↪  string_of_expr el) ^ ")"
81    | MatIndex(mat, e2, e3) ->
82      mat ^ "[" ^ string_of_expr e2 ^ "," ^ string_of_expr e3 ^ "]"
83    | MatIndexAssign(mat, e2, e3, e4) ->
84      mat ^ "[" ^ string_of_expr e2 ^ "," ^ string_of_expr e3 ^ "]"
85      ^ " = " ^ string_of_expr e4
86    | StructArrayAccess(strct, member, idx) -> strct ^ "." ^ member ^ "[" ^
      ↪  string_of_expr idx ^ "]"
87    | StructArrayAssign(strct, member, idx, e) ->
88      strct ^ "." ^ member ^ "[" ^ string_of_expr idx ^ "] = " ^ string_of_expr e
89
90    (* | StructLit(typ, bind_list) -> ignore(bind_list); string_of_typ typ (*
      ↪  TODO: make this real lol *) *)
91  let rec string_of_stmt = function
92      Block(stmts) ->
93        "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
94    | Expr(expr) -> string_of_expr expr ^ ";\n";
95    | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
96    | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
97    | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
98        string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
99    | For(e1, e2, e3, s) ->
100       "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
101       string_of_expr e3  ^ ") " ^ string_of_stmt s
102   | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
103
104
105 let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^ ";\n"
106
107 let string_of_fdecl fdecl = match fdecl.location with
108     Local -> string_of_typ fdecl.typ ^ " " ^
109       fdecl.fname ^ "(" ^ String.concat ", " (List.map snd fdecl.formals) ^
110       ")\n{\n" ^ String.concat "" (List.map string_of_vdecl fdecl.locals) ^
111       String.concat "" (List.map string_of_stmt fdecl.body) ^ "}\n"
112   | External -> "extern" ^ string_of_typ fdecl.typ ^ " " ^ fdecl.fname ^
```

```
      "(" ^ String.concat ", " (List.map snd fdecl.formals) ^ ");\n"

let string_of_struct_decl s =
  let vdecls = String.concat "" (List.map string_of_vdecl s.members) in
  "struct " ^ s.name ^ "{\n" ^
    vdecls ^
  "}\n"

let string_of_program program =
  let vars = program.global_vars
  and funcs = program.functions
  and structs = program.structs in
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "" (List.map string_of_struct_decl structs) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
(*============================================================================ *)


(*============================Semantic Type Checkers ====================== *)
let check_not_void exceptf = function
    (PrimitiveType(t), n) when t = Void -> raise (Failure (exceptf n))
  | _ -> ()

let check_no_structs exceptf = function
    (StructType(n), _) -> raise (Failure (exceptf n))
  | _ -> ()

let rec check_asn_silent lvaluet rvaluet =
  match (lvaluet, rvaluet) with
      (PrimitiveType(p1), PrimitiveType(p2)) -> if p1 = p2 then true else false
    | (StructType(s1), StructType(s2)) -> if s1 = s2 then true else
        (print_endline (s1 ^ s2); false)
    | (ArrayType(typ1), ArrayType(typ2)) ->
        if check_asn_silent typ1 typ2 then true else false
    | (FptrType(fp1), FptrType(fp2)) ->
        if List.length fp1 != List.length fp2 then
          (print_endline (string_of_typ (FptrType(fp1)) ^ string_of_typ
          ↪  (FptrType(fp2))); false)
        else if fp1 = fp2 then true else
          (print_endline (string_of_typ (FptrType(fp1)) ^ string_of_typ
          ↪  (FptrType(fp2))); false)
    | _ -> false

(* Raise an exception of the given rvalue type cannot be assigned to
   the given lvalue type *)
   (* TODO: pattern match on typ type *)
let check_assign lvaluet rvaluet expr =
```

```
158   if check_asn_silent lvaluet rvaluet then lvaluet else raise
159   (Failure ("illegal assignment " ^ string_of_typ lvaluet ^
160             " = " ^ string_of_typ rvaluet ^ " in " ^
161             string_of_expr expr))
162
163 let check_func_param_assign lvaluet rvaluet err =
164   if check_asn_silent lvaluet rvaluet then lvaluet else raise err
165
166 let rec contains x = function
167     [] -> false
168   | hd :: tl -> if x = hd then true else contains x tl
169
170 let rec try_get x = function
171     [] -> None
172   | hd :: tl -> if x = hd then Some x else try_get x tl
173
174 let match_primitive primitives = function
175     PrimitiveType(p) -> contains p (Array.to_list primitives)
176   | _ -> false
177
178 let match_struct = function
179     StructType(_) -> true
180   | _ -> false
181
182 let match_array = function
183     ArrayType(_) -> true
184   | _ -> false
185
186 (*============================== List Checkers ============================== *)
187 let report_duplicate exceptf lst =
188   let rec helper = function
189       n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
190     | _ :: t -> helper t
191     | [] -> ()
192   in helper (List.sort compare lst)
193
194 let rec get_last = function
195   [t] -> t
196 | _ :: tl -> get_last tl
197 | _ -> raise (Failure "must be nonempty list")
198
199 (*============================== Struct Checkers ============================ *)
200 let check_struct_not_empty exceptf = function
201     { name = n; members = []; } -> raise (Failure (exceptf n))
202   | _ -> ()
203
204 let check_struct_no_nested exceptf struct_decl =
```

```
205   let n = struct_decl.name in
206   if
207     List.exists match_struct (List.map (fun member -> fst member)
        ↪  struct_decl.members)
208   then raise (Failure (exceptf n)) else ()
209
210 let check_no_opaque exceptf = function
211     (_, n) -> raise (Failure (exceptf n))
212
213 let get_struct_member_type struct_decl member exceptf =
214   try
215     let member_bind = List.find (fun (_, n) -> n = member) struct_decl.members
216     in fst member_bind  (* return the typ *)
217   with Not_found -> raise (Failure exceptf)
218
219 let get_struct_member_idx struct_decl member =
220   let rec find idx = function
221       [] -> raise Not_found
222     | (_,name) :: tl -> if name = member then idx else find (idx+1) tl
223   in
224   try find 0 struct_decl.members
225   with Not_found -> raise (Failure (member ^ "not found in struct " ^
       ↪  struct_decl.name))
226
227 let is_struct_access s =
228   try
229     ignore (Str.search_forward (Str.regexp "[.]") s 0); true
230   with Not_found -> false
231
232 let parse_struct_access s =
233   let l = Str.split (Str.regexp "[.]") s in
234   let a = Array.of_list l in
235   (a.(0), a.(1))
236
237 (* foo.bar(), converts bar to __foo_bar(foo) *)
238 let methodify mthd_name s_name = "__" ^ s_name ^ "_" ^ mthd_name
239
240 (*============================== Array Checkers ============================== *)
241 let check_array_or_throw typ a_name =
242   if match_array typ then () else raise (Failure (a_name ^ " is not an array"))
243
244 let get_array_type = function
245     ArrayType(typ) -> typ
246   | _ -> raise (Failure "invalid array type")
247
248 (*============================== Function Checkers ============================ *)
249 let get_result_primitive_name f_name = function
```

```
    Void -> ""
  | _ -> f_name ^ "_result"
let get_result_name f_name = function
    PrimitiveType(t) -> get_result_primitive_name f_name t
  | _ -> f_name ^ "_result"

let parse_fptr_type typ_list =
  let arg_typs = if List.length typ_list = 1 then [] else
    (List.rev (List.tl (List.rev typ_list)))
  in
  let ret_typ = get_last typ_list in
  (arg_typs, ret_typ)

(*================================== Misc=================================== *)
let try_get_id_str = function
  Id(s) -> Some s
| _ -> None
```

9.3 Demo files

test-eigen-mnist.cqm

```
int main()
{
  int ret;
  fmatrix[] dst_fm_images;
  fmatrix[] dst_fm_labels;
  fmatrix fm;

  dst_fm_images = make(fmatrix, 50000);
  dst_fm_labels = make(fmatrix, 50000);

  ret = load_mnist_data(dst_fm_images, dst_fm_labels,
    "mnist_data/train-images-idx3-ubyte",
    "mnist_data/train-labels-idx1-ubyte"
  );

  // ret = load_mnist_data(dst_fm_images, dst_fm_labels,
  //   "mnist_data/t10k-images-idx3-ubyte",
  //   "mnist_data/t10k-labels-idx1-ubyte"
  // );

  printf("returned with value: %d\n", ret);

```

```
  print_mnist_image(dst_fm_images[49999]);
  print_mat(dst_fm_labels[49999]);

  return 0;
}
```

Makefile (mnist)

```
COMPILE = ./compile
PRELUDE = ./prelude

.PHONY: all
all:
	$(PRELUDE)
	$(COMPILE) mnist.cqm
	$(COMPILE) test-eigen-mnist.cqm

.PHONY: clean
clean:
	rm *.s *.ll *.so *.o
	rm mnist test-eigen-mnist
```

9.4 Library files

deep.cqm

```
extern int load_mnist_data(
  fmatrix[] dst_fm_images,
  fmatrix[] dst_fm_labels,
  string image_filename,
  string label_filename
);

struct fc_model {
  fmatrix[] train_x;
  fmatrix[] train_y;
  fmatrix[] test_x;
  fmatrix[] test_y;
  fmatrix[] biases;
  fmatrix[] weights;
  int[] layer_sizes;
  int epochs;
```

```
  int mini_batch_size;
  float learning_rate;
  fp (float) weight_init;
  fp (float, float) activate;
  fp (float, float) activate_prime;
  fp (fmatrix, fmatrix, float) cost;
  fp (fmatrix, fmatrix, fmatrix, fmatrix) cost_prime;
}

struct backprop_deltas {
  fmatrix[] weight_deltas;
  fmatrix[] bias_deltas;
}

[struct backprop_deltas bpds] free() void {
  free_fmat_arr(bpds.weight_deltas);
  free_fmat_arr(bpds.bias_deltas);
  free(bpds);
}

[struct fc_model fc] free_model() void {
  free_fmat_arr(fc.train_x);
  free_fmat_arr(fc.train_y);
  free_fmat_arr(fc.test_x);
  free_fmat_arr(fc.test_y);
  free_fmat_arr(fc.biases);
  free_fmat_arr(fc.weights);
  return;
}

[struct fc_model fc] init_biases() void {
  int i;
  fc.biases = make(fmatrix, len(fc.layer_sizes) - 1);
  for (i = 1; i < len(fc.layer_sizes); i = i + 1) {
    fc.biases[i-1] = init_fmat_zero(fc.layer_sizes[i], 1);
    fc.biases[i-1] = populate_fmat(fc.biases[i-1], fc.weight_init);
  }
  return;
}

/* construct weight matrices and initialize with random values */
[struct fc_model fc] init_weights() void {
  int: i, input_nodes, r, c;
  fmatrix fm;
  float stdv;
  fc.weights = make(fmatrix, len(fc.layer_sizes) - 1);
  for (i = 1; i < len(fc.layer_sizes); i = i + 1) {
```

```
    fc.weights[i-1] = init_fmat_zero(fc.layer_sizes[i], fc.layer_sizes[i-1]);
    fc.weights[i-1] = populate_fmat(fc.weights[i-1], fc.weight_init);
    input_nodes = fc.layer_sizes[i-1];
    stdv = sqrt(float_of_int(input_nodes));
    fm = fc.weights[i-1];
    /* normalize weights wrt input edges */
    for (r = 0; r < rows(fm); r = r + 1) {
      for (c = 0; c < cols(fm); c = c + 1) {
        fm[r,c] = fm[r,c] / stdv;
      }
    }
  }
  return;
}

[struct fc_model fc] predict(fmatrix X) fmatrix {
  int i;
  fmatrix: fm, tmp1, tmp2;
  for (i = 0; i < len(fc.weights); i = i + 1) {
    tmp1 = fc.weights[i] .. X;
    tmp2 = tmp1 + fc.biases[i];
    X = tmp2 => map(fc.activate);
    free(tmp1); free(tmp2);
  }
  return X;
}

/* Zeros the update matrices for next minibatch */
[struct fc_model fc] zero_deltas(
  fmatrix[] weights, fmatrix[] biases, bool should_free) void {
  int i;
  for (i = 0; i < len(fc.weights); i = i + 1) {
    if (should_free) {
      free(weights[i]);
      free(biases[i]);
    }
    weights[i] = init_fmat_zero(
      rows(fc.weights[i]), cols(fc.weights[i])
    );
    biases[i] = init_fmat_zero(
      rows(fc.biases[i]), cols(fc.biases[i])
    );
  }
}

[struct fc_model fc] train() void {
  bool should_free;
```

```
  int: mini, e, i, train_idx, train_size, l, ratio, progress;
  struct backprop_deltas bpds;
  fmatrix[]: sum_weight_deltas, sum_bias_deltas;
  fmatrix: tmp1, tmp2, tmp3, tmp4;
  int[] train_indices;

  /* initialize training indices */
  train_size = len(fc.train_x);
  train_indices = make(int, train_size);
  for (i = 0; i < train_size; i = i + 1) {
    train_indices[i] = i;
  }
  ratio = (train_size / 30);

  /* initialize parameters and update matrices */
  fc.init_weights();
  fc.init_biases();
  sum_weight_deltas = make(fmatrix, len(fc.weights));
  sum_bias_deltas = make(fmatrix, len(fc.biases));
  should_free = false;

  /* initialize delta matrices */
  bpds = make(struct backprop_deltas);
  bpds.weight_deltas = make(fmatrix, len(fc.weights));
  bpds.bias_deltas = make(fmatrix, len(fc.biases));

  printf("Performance Before Training:\n");
  for (e = 0; e < fc.epochs; e = e + 1) {
    fc.evaluate();
    printf("Training Epoch %d: [", e+1);
    train_idx = 0;
    progress = 0;
    shuffle(train_indices);

    while (train_idx < train_size) {
      fc.zero_deltas(sum_weight_deltas, sum_bias_deltas, should_free);
      /* accumulate over mini patch */

      for (
        mini = train_idx;
        mini < min_int(train_idx + fc.mini_batch_size, train_size);
        mini = mini + 1
      ) {

        i = train_indices[mini];
        fc.backprop(fc.train_x[i], fc.train_y[i], bpds, should_free);
        for (l = 0; l < len(sum_weight_deltas); l = l+1) {
```

```
          tmp1 = sum_weight_deltas[l];
          tmp2 = sum_bias_deltas[l];
          sum_weight_deltas[l] = sum_weight_deltas[l] + bpds.weight_deltas[l];
          sum_bias_deltas[l] = sum_bias_deltas[l] + bpds.bias_deltas[l];
          free(tmp1);
          free(tmp2);
        }
        should_free = true;  // everything has been initialized once, now free

      }

      /* update network weights */
      for (l = 0; l < len(sum_weight_deltas); l = l + 1) {
        /* normalize */
        tmp1 = sum_weight_deltas[l];
        tmp2 = sum_bias_deltas[l];
        sum_weight_deltas[l] =
          sum_weight_deltas[l] * (fc.learning_rate /
           ↪  float_of_int(fc.mini_batch_size));
        sum_bias_deltas[l] =
          sum_bias_deltas[l] * (fc.learning_rate /
           ↪  float_of_int(fc.mini_batch_size));
        free(tmp1);
        free(tmp2);
        tmp3 = fc.weights[l];
        tmp4 = fc.biases[l];
        fc.weights[l] = fc.weights[l] - sum_weight_deltas[l];
        fc.biases[l] = fc.biases[l] - sum_bias_deltas[l];
        free(tmp3); free(tmp4);
      }
      train_idx = train_idx + fc.mini_batch_size;
      if (train_idx > progress) {
        printf("=");
        flush();
        progress = progress + ratio;
      }
    }
    printf("]\n");
  }
  fc.evaluate();  // final performance

  return;
}

[struct fc_model fc] backprop(
  fmatrix x, fmatrix y, struct backprop_deltas bpds, bool should_free
) void {
```

```
int: i, num_param_layers;
fmatrix: activation, z, z_prime, delta, actv_transpose, tmp1, tmp2, tmp3,
 ↪  tmp4;
fmatrix[]: activations, zs;
fp (float, float) activate_prime;
fp (fmatrix, fmatrix, fmatrix, fmatrix) cost_prime;

num_param_layers = len(fc.weights);

activate_prime = fc.activate_prime;
cost_prime = fc.cost_prime;

activations = make(fmatrix, len(fc.layer_sizes));
activations[0] = copy(x);
zs = make(fmatrix, num_param_layers);

// free from last run
for (i = 0; i < len(fc.weights); i = i + 1) {
  if (should_free) {
    free(bpds.weight_deltas[i]);
    free(bpds.bias_deltas[i]);
  }
}

// forward pass
for (i = 0; i < len(fc.weights); i = i + 1) {
  // tmp1 = (fc.weights[i] .. activation);
  tmp1 = (fc.weights[i] .. activations[i]);
  z = tmp1 + fc.biases[i];
  free(tmp1);
  zs[i] = z;
  activations[i+1] = map(z, fc.activate);
}

// TODO: cannot distinguish calling fp from method dispatch.
// backward pass

tmp1 = map(zs[len(zs)-1], activate_prime);
delta = cost_prime(tmp1, activations[len(activations) - 1], y);
free(tmp1);

// free(activations[len(activations) - 1]);
bpds.bias_deltas[num_param_layers - 1] = delta;
tmp1 = ((activations[len(activations) - 2])^);
bpds.weight_deltas[num_param_layers - 1] = delta .. tmp1;
free(tmp1);

```

```
249
250   for (i = 2; i < len(fc.layer_sizes); i = i + 1) {
251     z = zs[len(zs)-i];
252     z_prime = map(z, activate_prime);
253     tmp1 = (fc.weights[num_param_layers - i + 1])^;
254     tmp2 = tmp1 .. bpds.bias_deltas[num_param_layers - i + 1];
255     bpds.bias_deltas[num_param_layers - i] = tmp2 * z_prime;
256     free(z_prime); free(tmp1); free(tmp2);
257     // ------------The Leak----------
258     tmp1 = (activations[len(activations) - i - 1]^);
259     bpds.weight_deltas[num_param_layers - i] =
260       bpds.bias_deltas[num_param_layers - i] .. tmp1;
261     free(tmp1);
262     // ------------------------------
263   }
264
265   free_fmat_arr(activations);
266   free_fmat_arr(zs);
267
268   return;
269 }
270
271 [struct fc_model fc] evaluate() void {
272   int: i, pred, correct, gold;
273   fp (fmatrix, fmatrix, float) cost_func;
274   float cost;
275   fmatrix out;
276
277   cost_func = fc.cost;
278
279   cost = 0.;
280   correct = 0;
281   for (i = 0; i < len(fc.test_x); i = i + 1) {
282     out = fc.predict(fc.test_x[i]);
283     pred = argmax(out);
284     gold = argmax(fc.test_y[i]);
285     cost = cost + cost_func(out, fc.test_y[i]);
286     free(out);
287
288     if (pred == gold) {
289       correct = correct + 1;
290     }
291   }
292   printf("\ttest set cost: %f\n", cost);
293   printf("\ttest set accuracy: %d/%d = %f\n", correct, len(fc.test_x),
    ↪  float_of_int(correct) / float_of_int(len(fc.test_x)));
294 }
```

```
295
296 [struct fc_model fc] demo(int num) void {
297   int: i, closest_guess, gold, pred;
298   float: min_cost, cost;
299   fmatrix out;
300   fp (fmatrix, fmatrix, float) cost_func;
301   int[]: correct_indices, incorrect_indices;
302
303   cost_func = fc.cost;
304   min_cost = -1.;
305   cost = 0.;
306
307   correct_indices = make(int, 0);
308   incorrect_indices = make(int, 0);
309
310   for (i = 0; i < len(fc.test_x); i = i + 1) {
311     out = fc.predict(fc.test_x[i]);
312     pred = argmax(out);
313     gold = argmax(fc.test_y[i]);
314
315     cost = cost_func(out, fc.test_y[i]);
316     free(out);
317
318     if (pred == gold) {
319       correct_indices = append(correct_indices, i);
320     } else {
321       if (cost < min_cost || min_cost < 0.) {
322         min_cost = cost;
323         closest_guess = i;
324       } else {
325         incorrect_indices = append(incorrect_indices, i);
326       }
327     }
328   }
329
330   printf("================Correct Guesses===============\n");
331   for (i = 0; i < min_int(len(correct_indices), num); i = i + 1) {
332       print_mnist_image(fc.test_x[correct_indices[i]]);
333       printf("Prediction: %d\n", argmax(fc.test_y[correct_indices[i]]));
334   }
335
336   printf("\n================Incorrect Guesses===============\n");
337   incorrect_indices[0] = closest_guess;
338   for (i = 0; i < min_int(len(incorrect_indices), num); i = i + 1) {
339       print_mnist_image(fc.test_x[incorrect_indices[i]]);
340       out = fc.predict(fc.test_x[incorrect_indices[i]]);
341       pred = argmax(out);
```

```
342       free(out);
343       printf("Prediction: %d\n", pred);
344     }
345
346     return;
347   }
348
349   void print_mnist_image(fmatrix fm) {
350     int: i, j, idx;
351     float val;
352     for (i = 0; i < 28; i = i + 1) {
353       for (j = 0; j < 28; j = j + 1) {
354         idx = 28*i + j;
355         val = fm[idx, 0];
356         if (val > 0.50) {
357           printf("[0]");
358         } else {
359           if (val > 0.01 ) {
360             printf("[|]");
361           } else {
362             printf("[ ]");
363           }
364         }
365         if (j == 27) {
366           print_line();
367         }
368       }
369     }
370   }
371
372   /* Deep-learning related math functions */
373   int argmax(fmatrix fm) {
374     int r;
375     int m;
376     m = 0;
377     for (r = 0; r < rows(fm); r = r + 1) {
378       if (fm[r,0] > fm[m,0]) {
379         m = r;
380       }
381     }
382     return m;
383   }
384
385   float l2_norm(fmatrix fm) {
386     int r;
387     float acc;
388     for (r = 0; r < rows(fm); r = r + 1) {
```

```
    acc = acc + square(fm[r,0]);
  }
  return sqrt(acc);
}

float quadratic_cost(fmatrix x, fmatrix y) {
  float ret;
  fmatrix fm;
  fm = x - y;
  ret = square(l2_norm(fm)) * .5;
  free(fm);
  return ret;
}

fmatrix quadratic_cost_prime(fmatrix z, fmatrix x, fmatrix y) {
  fmatrix: tmp1, tmp2;
  tmp1 = (x - y);
  tmp2 = tmp1 * z;
  free(tmp1);
  return tmp2;
}

float mat_sum(fmatrix fm) {
  int r;
  float acc;
  acc = 0.;
  for (r = 0; r < rows(fm); r = r + 1) {
    if (fm[r,0] > -1000.) {
        acc = acc + fm[r,0];
    }
  }
  return acc;
}

/* ssshhh this is legit, ok? */
float cross_entropy_cost(fmatrix a, fmatrix y) {
  return quadratic_cost(a, y);
  // fmatrix: tmp1, tmp2, tmp3, lhs, rhs, ret;
  // float sum;
  // sum = 0.;
  // tmp1 = map(a, log);
  // tmp2 = a-1.;
  // tmp3 = -y;
  // lhs = tmp3 * tmp1;
  // free(tmp3); free(tmp1);
  // tmp1 = map(tmp2, log);
  // free(tmp2);
```

```
  // tmp3 = (1.-y);
  // rhs = tmp3 * tmp1;
  // free(tmp3); free(tmp1);
  // ret = lhs - rhs;
  // free(lhs); free(rhs);
  // sum = mat_sum(ret);
  // free(ret);
  // return sum;
}

fmatrix cross_entropy_cost_prime(fmatrix z, fmatrix x, fmatrix y) {
  return (x-y);
}

float sigmoid(float z) {
    return 1.0 / (1.0 + exp(-z));
}

float sigmoid_prime(float z) {
  return sigmoid(z) * (1. - sigmoid(z));
}

float norm_init() {
  return rand_norm(0., 1.);
}

/* for relu */
float const_init() {
  return .2;
}

float tanh_prime(float z) {
  return (1. - square(tanh(z)));
}

float relu(float z) {
  return max(0., z);
}

float relu_prime(float z) {
  if (z >= 0.) {
    return 1.;
  }
  return 0.;
}
```

eigen.cqm

```
extern void print_mat(fmatrix fm);

extern float mat_index(fmatrix fm, int i, int j);
extern float mat_index_assign(fmatrix fm, int i, int j, float f);

extern int rows(fmatrix fm);
extern int cols(fmatrix fm);
extern fmatrix init_fmat_zero(int r, int c);
extern fmatrix init_fmat_const(float s, int r, int c);
extern fmatrix init_fmat_identity(int r, int c);
extern fmatrix arr_to_fmat(float[] arr, int t, int c);

extern fmatrix copy(fmatrix fm);
extern void del_mat(fmatrix fm);
extern fmatrix map(fmatrix fm, fp (float, float) f_ptr);

extern fmatrix mm_add(fmatrix fm1, fmatrix fm2);
extern fmatrix mm_sub(fmatrix fm1, fmatrix fm2);
extern fmatrix mm_mult(fmatrix fm1, fmatrix fm2);

extern fmatrix mm_div(fmatrix fm1, fmatrix fm2);
extern fmatrix dot(fmatrix fm1, fmatrix fm2);

extern fmatrix sm_add(fmatrix fm, float s, int rev);
extern fmatrix sm_sub(fmatrix fm, float s, int rev);
extern fmatrix sm_mult(fmatrix fm, float s, int rev);
extern fmatrix sm_div(fmatrix fm, float s, int rev);
// extern fmatrix sm_div(fmatrix fm, float s);

extern fmatrix smeq(fmatrix fm, float s);

extern fmatrix transpose(fmatrix fm);
extern fmatrix negate(fmatrix fm);

/*
populates each value by falling f()
Used for random initialization.
*/
fmatrix populate_fmat(fmatrix fm, fp (float) f) {
  int: i,j;
  for (i = 0; i < rows(fm); i = i + 1) {
    for (j = 0; j < cols(fm); j = j + 1) {
      fm[i,j] = f();
    }
  }
```

```
  return fm;
}

/* apply f to every element of fm */
fmatrix f_fmat(fmatrix fm, fp (float, float) f) {
  int: i,j;
  fmatrix fm1;
  fm1 = copy(fm);
  for (i = 0; i < rows(fm); i = i + 1) {
    for (j = 0; j < cols(fm); j = j + 1) {
      fm1[i,j] = f(fm[i,j]);
    }
  }
  return fm1;
}

void free_fmat_arr(fmatrix[] arr) {
  int i;
  for (i = 0; i < len(arr); i = i + 1) {
    free(arr[i]);
  }
  free_arr(arr);
}
```

eigen_mnist.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "eigen_test.h"

/* code copied and modified from https://github.com/projectgalateia/mnist/ */
/* to build: make eigen_mnist */

static unsigned int mnist_bin_to_int(char *v)
{
        int i;
        unsigned int ret = 0;

        for (i = 0; i < 4; ++i) {
                ret <<= 8;
                ret |= (unsigned char)v[i];
        }

        return ret;
```

```
}

int modulo(int x, int y) {
        return x % y;
}

void flush() {
        fflush(stdout);
}

int load_mnist_data(
  matrix_t *dst_fm_images,
  matrix_t *dst_fm_labels,
  char *image_filename,
  char *label_filename
) {
  int return_code = 0;
  int i;
  char tmp[4];
  matrix_t *src_fm_images;
  matrix_t *src_fm_labels;

  unsigned int image_cnt, label_cnt;
  unsigned int image_dim[2];

  FILE *ifp = fopen(image_filename, "rb");
  FILE *lfp = fopen(label_filename, "rb");

  if (!ifp || !lfp) {
    return_code = -1; /* No such files */
    goto cleanup;
  }

  fread(tmp, 1, 4, ifp);
  if (mnist_bin_to_int(tmp) != 2051) {
    return_code = -2; /* Not a valid image file */
    goto cleanup;
  }

  fread(tmp, 1, 4, lfp);
  if (mnist_bin_to_int(tmp) != 2049) {
    return_code = -3; /* Not a valid label file */
    goto cleanup;
  }

  fread(tmp, 1, 4, ifp);
  image_cnt = mnist_bin_to_int(tmp);
```

```
67        printf("images: %d\n", image_cnt);
68
69    fread(tmp, 1, 4, lfp);
70    label_cnt = mnist_bin_to_int(tmp);
71
72    if (image_cnt != label_cnt) {
73      return_code = -4; /* Element counts of 2 files mismatch */
74      goto cleanup;
75    }
76
77    for (i = 0; i < 2; ++i) {
78      fread(tmp, 1, 4, ifp);
79      image_dim[i] = mnist_bin_to_int(tmp);
80    }
81
82    if (image_dim[0] != 28 || image_dim[1] != 28) {
83      return_code = -2; /* Not a valid image file */
84      goto cleanup;
85    }
86
87    src_fm_images = (matrix_t *)malloc(sizeof(matrix_t) * image_cnt);
88    src_fm_labels = (matrix_t *)malloc(sizeof(matrix_t) * label_cnt);
89
90    for (i = 0; i < image_cnt; ++i) {
91      int j;
92      unsigned char read_data[28 * 28];
93
94      src_fm_images[i] = init_fmat_zero(28*28, 1);
95      fread(read_data, 1, 28*28, ifp);
96      for (j = 0; j < 28*28; ++j) {
97        mat_index_assign(src_fm_images[i], j, 0, read_data[j] / 255.0);
98      }
99
100     src_fm_labels[i] = init_fmat_zero(10, 1);
101     fread(tmp, 1, 1, lfp);
102     mat_index_assign(src_fm_labels[i], tmp[0], 0, 1.0);
103     // printf("label: %d\n", tmp[0]);
104   }
105
106   memcpy(dst_fm_images, src_fm_images, sizeof(matrix_t) * image_cnt);
107   memcpy(dst_fm_labels, src_fm_labels, sizeof(matrix_t) * label_cnt);
108
109       free(src_fm_images);
110       free(src_fm_labels);
111
112       printf("Successfully read image file: %s\n", image_filename);
113       printf("Successfully read label file: %s\n", label_filename);
```

```
114         flush();
115
116   cleanup:
117   if (ifp) fclose(ifp);
118   if (lfp) fclose(lfp);
119
120   return return_code;
121 }
122
123
124 /* DO NOT DELETE! this is effecitvely a unit test for load_mnist_data */
125 /*
126 int main()
127 {
128   matrix_t *dst_fm_images;
129   matrix_t *dst_fm_labels;
130   int ret;
131
132   dst_fm_images = (matrix_t *)malloc(sizeof(matrix_t) * 50000);
133   dst_fm_labels = (matrix_t *)malloc(sizeof(matrix_t) * 50000);
134
135   // ret = load_mnist_data(dst_fm_images, dst_fm_labels,
136   //   "mnist_data/train-images-idx3-ubyte",
137   //   "mnist_data/train-labels-idx1-ubyte"
138   // );
139
140   ret = load_mnist_data(dst_fm_images, dst_fm_labels,
141     "mnist_data/t10k-images-idx3-ubyte",
142     "mnist_data/t10k-labels-idx1-ubyte"
143   );
144
145   printf("returned with value: %d\n", ret);
146
147   int idx;
148   double val;
149   for (int i = 0; i < 28; ++i) {
150     for (int j = 0; j < 28; ++j) {
151       idx = 28*i + j;
152       val = mat_index(dst_fm_images[9000], idx, 0);
153       if (val > 0.01)
154         printf("[0]");
155       else
156         printf("[ ]");
157       if (j == 27)
158         printf("\n");
159     }
160   }
```

```
161
162   // print_mat(dst_fm_images[0]);
163   print_mat(dst_fm_labels[9000]);
164
165   return 0;
166 }
167 */
```

eigen_test.c

```
1  #include <stdio.h>
2  #include "eigen_test.h"
3
4  int main(){
5          matrix_t tmp = init_fmat_identity(4, 4);
6          matrix_t tmp2 = init_fmat_const(1, 4, 4);
7          matrix_t tmp3 = init_fmat_const(2, 4, 4);
8
9          matrix_t tmp4 = mm_add(tmp2, tmp3);
10         matrix_t tmp5 = dot(tmp2, tmp3);
11         matrix_t tmp6 = sm_div(tmp, 2, 0);
12
13         printf("%f\n", mat_index(tmp3, 1,1));
14         printf("%f\n", mat_index(tmp2, 1,1));
15
16         print_mat(tmp);
17         print_mat(tmp2);
18         print_mat(tmp3);
19         print_mat(tmp4);
20         print_mat(tmp5);
21
22         mat_index_assign(tmp, 1, 1, 3.14);
23         mat_index_assign(tmp, 2, 1, 3.14);
24         print_mat(tmp);
25
26         del_mat(tmp);
27         del_mat(tmp2);
28         del_mat(tmp3);
29         del_mat(tmp4);
30         del_mat(tmp5);
31 }
```

eigen_test.h

```c
#ifdef __cplusplus
#include <iostream>
#include <Eigen/Dense>
extern "C" {
#endif

typedef void * matrix_t;

/* =============== utility functions =============== */

void print_mat(matrix_t);
void onion_matrix_test();

int rows(matrix_t);
int cols(matrix_t);

matrix_t init_fmat_zero(int, int);
matrix_t init_fmat_const(double, int, int);
matrix_t init_fmat_identity(int, int);
matrix_t init_fmat_literal(double *, int, int);
matrix_t arr_to_fmat(double *, int, int);
matrix_t map(matrix_t, double (*f_ptr)(double));
matrix_t copy(matrix_t);

void del_mat(matrix_t);

/* =============== Index and Slicing =============== */

double mat_index(matrix_t, int, int);
double mat_index_assign(matrix_t, int, int, double);

/* =============== Binary Operations =============== */

// Matrix-Matrix operations
matrix_t mm_add(matrix_t, matrix_t);
matrix_t mm_sub(matrix_t, matrix_t);
matrix_t mm_mult(matrix_t, matrix_t);
matrix_t mm_div(matrix_t, matrix_t);
matrix_t dot(matrix_t, matrix_t);

// Scalar-Matrix operations
matrix_t sm_add(matrix_t, double, int);
matrix_t sm_sub(matrix_t, double, int);
matrix_t sm_mult(matrix_t, double, int);
matrix_t sm_div(matrix_t, double, int);
// matrix_t sm_div(matrix_t, double);
matrix_t smeq(matrix_t, double);
```

```
48
49  /* ============== Matrix Unary Operations ============== */
50  matrix_t transpose(matrix_t);
51  matrix_t negate(matrix_t);
52
53  #ifdef __cplusplus
54  }
55  #endif
```

eigen_test_lib.cpp

```
1   #include "eigen_test.h"
2
3
4
5   using namespace Eigen;
6
7   typedef Matrix<double, Dynamic, Dynamic, RowMajor> MatrixXdr;
8   /* ============================ Utility Functions ============================
    ↪   */
9
10  void onion_matrix_test(){
11          MatrixXdr tmp_m = MatrixXdr::Constant(5, 5, 2.4);
12          std::cout << tmp_m << std::endl;
13  }
14
15  MatrixXdr* mat_cast(matrix_t undef_mptr){
16          return static_cast<MatrixXdr*>(undef_mptr);
17  }
18
19  void print_mat(matrix_t undef_mptr){
20          MatrixXdr* def_mptr = mat_cast(undef_mptr);
21          std::cout << *def_mptr << std::endl;
22  }
23
24  int rows(matrix_t undef_mptr){
25          MatrixXdr* def_mptr = mat_cast(undef_mptr);
26          return (*def_mptr).rows();
27  }
28
29  int cols(matrix_t undef_mptr){
30          MatrixXdr* def_mptr = mat_cast(undef_mptr);
31          return (*def_mptr).cols();
32  }
33
```

```
matrix_t map(matrix_t undef_mptr, double (*f_ptr)(double)){
        MatrixXdr* def_mptr = mat_cast(undef_mptr);
        MatrixXdr* tmp_mptr = new MatrixXdr;
        *tmp_mptr =  (*def_mptr).unaryExpr(f_ptr);
        return tmp_mptr;


}

/* ============================ Index and Slicing ============================== */

double mat_index(matrix_t undef_mptr, const int r, const int c) {
        MatrixXdr* def_mptr = mat_cast(undef_mptr);
        return (*def_mptr)(r,c);
}

double mat_index_assign(matrix_t undef_mptr, int r, int c, double f) {
        MatrixXdr* def_mptr = mat_cast(undef_mptr);
        (*def_mptr)(r,c) = f;
        return f;
}

/* ============================ Matrix Inititialization
    ============================== */

MatrixXdr* init_fmat(const int d1, const int d2, const double c, const int
    op_id){
        MatrixXdr* tmp_mptr = new MatrixXdr;
        switch (op_id) {
                case 0: *tmp_mptr = MatrixXdr::Zero(d1, d2); break;
                case 1: *tmp_mptr = MatrixXdr::Constant(d1, d2, c); break;
                case 2: *tmp_mptr = MatrixXdr::Identity(d2, d2); break;
        }

        return tmp_mptr;
}

matrix_t init_fmat_zero(const int d1, const int
    d2)                                                 {return init_fmat(d1,
    d2, -1, 0);}
matrix_t init_fmat_const(const double c, const int d1, const int
    d2)        {return init_fmat(d1, d2, c, 1);}
matrix_t init_fmat_identity(const int d1, const int
    d2)                                         {return init_fmat(d1, d2, -1,
    2);}
matrix_t init_fmat_literal(double * arr, const int d1, const int d2)          {
```

```
73                  MatrixXdr* tmp_mptr = new MatrixXdr;
74                  (*tmp_mptr) = Map<MatrixXdr>(arr, d1, d2);
75                  return tmp_mptr;
76  }
77
78  matrix_t arr_to_fmat(double * arr, const int d1, const int d2) {
79                  MatrixXdr* tmp_mptr = new MatrixXdr;
80                  (*tmp_mptr) = Map<MatrixXdr>(arr, d1, d2);
81                  return tmp_mptr;
82  }
83
84  matrix_t copy(matrix_t undef_mptr){
85          MatrixXdr* tmp_mptr = new MatrixXdr;
86          MatrixXdr* def_mptr = mat_cast(undef_mptr);
87          *tmp_mptr = *def_mptr;
88          return tmp_mptr;
89  }
90
91  void del_mat(matrix_t undef_mptr){
92          MatrixXdr *  def_ptr = mat_cast(undef_mptr);
93          delete def_ptr;
94  }
95
96
97  /* ============================ Matrix Binary Operations
    ↪   ============================== */
98
99  MatrixXdr* binary_operations(matrix_t undef_mptr1, matrix_t undef_mptr2, double
    ↪   scalar, int op_id){
100         MatrixXdr* def_mptr1 = mat_cast(undef_mptr1);
101         MatrixXdr* def_mptr2 = mat_cast(undef_mptr2);
102         MatrixXdr* tmp_mptr  = new MatrixXdr;
103
104         switch(op_id) {
105
106                 /* ======================== Matrix Matrix Operations
                    ↪   ========================== */
107                 // matrix-matrix addition
108                 case 0: *tmp_mptr = *def_mptr1 + *def_mptr2; break;
109                 // matrix-matrix subtraction
110                 case 1: *tmp_mptr = *def_mptr1 - *def_mptr2; break;
111                 // matrix-matrix multiplication
112                 case 2: *tmp_mptr = (*def_mptr1).cwiseProduct(*def_mptr2);
                    ↪   break;
113                 // matrix-matrix division
114                 case 3: *tmp_mptr = (*def_mptr1).cwiseQuotient(*def_mptr2);
                    ↪   break;
```

```
115                 // matrix-matrix dot product
116                 case 4: (*tmp_mptr).noalias() = ((*def_mptr1) * (*def_mptr2));
                     ↪  break;
117
118                 /* ======================== Scalar Matrix Operations
                     ↪  ======================== */
119
120                 // scalar matrix addition
121                 case 5: *tmp_mptr = (*def_mptr1).array() + scalar; break;
122                 // scalar matrix subtraction
123                 case 6: *tmp_mptr = (*def_mptr1).array() - scalar; break;
124                 case 7: *tmp_mptr = scalar - (*def_mptr1).array(); break;
125                 // scalar matrix multiplication
126                 case 8: *tmp_mptr = *def_mptr1 * scalar; break;
127                 // sclar matrix division
128                 case 9: *tmp_mptr = *def_mptr1 / scalar; break;
129                 case 10: *tmp_mptr = scalar  * ((*def_mptr1).cwiseInverse());
                     ↪  break;
130         }
131
132         return tmp_mptr;
133 }
134
135 MatrixXdr* binary_operations(matrix_t undef_mptr1, matrix_t undef_mptr2, int
     ↪  op_id){
136         return binary_operations(undef_mptr1, undef_mptr2, 0, op_id);
137 }
138
139 MatrixXdr* binary_operations(matrix_t undef_mptr, double scalar, int op_id){
140         MatrixXdr tmp_m;
141         return binary_operations(undef_mptr, &tmp_m, scalar, op_id);
142 }
143
144 matrix_t mm_add(matrix_t undef_mptr1, matrix_t undef_mptr2)  { return
     ↪  binary_operations(undef_mptr1, undef_mptr2, 0); }
145 matrix_t mm_sub(matrix_t undef_mptr1, matrix_t undef_mptr2)  { return
     ↪  binary_operations(undef_mptr1, undef_mptr2, 1); }
146 matrix_t mm_mult(matrix_t undef_mptr1, matrix_t undef_mptr2) { return
     ↪  binary_operations(undef_mptr1, undef_mptr2, 2); }
147 matrix_t mm_div(matrix_t undef_mptr1, matrix_t undef_mptr2)  { return
     ↪  binary_operations(undef_mptr1, undef_mptr2, 3); }
148 matrix_t dot(matrix_t undef_mptr1, matrix_t undef_mptr2)                {
     ↪  return binary_operations(undef_mptr1, undef_mptr2, 4); }
149
150 matrix_t sm_add(matrix_t undef_mptr, double s, int rev)
     ↪  { return binary_operations(undef_mptr, s, 5); }
151 matrix_t sm_sub(matrix_t undef_mptr, double s, int rev)
     ↪           { return rev ? binary_operations(undef_mptr, s, 6) :
```

```
matrix_t sm_mult(matrix_t undef_mptr, double s, int rev)                {
 ↪   return binary_operations(undef_mptr, s, 8); }
matrix_t sm_div(matrix_t undef_mptr, double s, int rev)
 ↪   { return rev ? binary_operations(undef_mptr, s, 9) :
```

```
// matrix_t sm_div(matrix_t undef_mptr, double s)        { return
 ↪   binary_operations(undef_mptr, s, 9); }
```

```
// matrix_t smeq(double s, matrix_t undef_mptr)        { return
 ↪   binary_operations(undef_mptr, MatrixXdr* tmp, s, 9); }
```

```
/* ============================== Matrix Unary Operations Operations
 ↪   ============================== */
```

```
matrix_t transpose(matrix_t undef_mptr){
        MatrixXdr* def_mptr = mat_cast(undef_mptr);
        MatrixXdr* tmp_mptr = new MatrixXdr;

        *tmp_mptr = (*def_mptr).transpose();
        return tmp_mptr;
}

matrix_t negate(matrix_t undef_mptr){
        return sm_mult(undef_mptr, -1, 0);
}
```

io.cqm

```
extern void flush(); // flush stdout
```

```

void print(int i) { printf("%d\n", i); }
void printb(bool b) { printf("%d\n", b); }
void print_float(float f) { printf("%f\n", f); }
void print_string(string s) { printf("%s\n", s); }
void print_line() { printf("\n"); }

void print_fmat_arr(fmatrix[] arr) {
  int i;
  for (i = 0; i < len(arr); i = i + 1) {
    print_mat(arr[i]);
  }
}

void print_fmat_arr_dims(fmatrix[] arr) {
  int i;
  for (i = 0; i < len(arr); i = i + 1) {
    printf("rows: %d cols: %d\n", rows(arr[i]), cols(arr[i]));
  }
}
```

math.cqm

```
/* Trig fns */
extern float sin(float x);
extern float cos(float x);
extern float tan(float x);
extern float sinh(float x);
extern float cosh(float x);
extern float tanh(float x);
extern float asin(float x);
extern float acos(float x);
extern float atan(float x);

extern float fabs(float x);
extern float exp(float x);
extern float log(float x);  // this is natural log
extern float log10(float x); // base 10 log
extern float pow(float x, float y);
extern int modulo(int x, int y);

extern int rand();
extern void srand(int seed);

float sqrt(float x) { return pow(x, 0.5); }
```

```
float square(float x) { return pow(x, 2.); }
float max(float x, float y) {
  if (x >= y) {
    return x;
  }
  return y;
}

int min_int(int x, int y) {
  if (x > y) {
    return y;
  }
  return x;
}

/*
sample from normal distribution using two uniform variables.
Code found at:
 ↪  https://phoxis.org/2013/05/04/generating-random-numbers-from-normal-distribution-in-c/
*/

float rand_norm(float mu, float sigma) {
  float: U1, U2, W, mult, X1, X2;
  float RAND_MAX;

  RAND_MAX = 2147483647.0;

  U1 = -1. + (float_of_int(rand()) / RAND_MAX) * 2.;
  U2 = -1. + (float_of_int(rand()) / RAND_MAX) * 2.;
  W = pow(U1, 2.) + pow(U2, 2.);

  while (W >= 1. || W == 0.) {
    U1 = -1. + (float_of_int(rand()) / RAND_MAX) * 2.;
    U2 = -1. + (float_of_int(rand()) / RAND_MAX) * 2.;
    W = pow(U1, 2.) + pow(U2, 2.);
  }

  mult = sqrt((-2. * log(W)) / W);
  X1 = U1 * mult;
  X2 = U2 * mult;

  return (mu + sigma * X1);
}

void shuffle(int[] arr) {
  int: i, j, k, n, RAND_MAX;
  RAND_MAX = 2147483647;
```

```
  n = len(arr);
  for (i = 0; i < n - 1; i = i + 1) {
    j = i + rand() / (RAND_MAX / (n - i) + 1);
    k = arr[j];
    arr[j] = arr[i];
    arr[i] = k;
  }
}
```

9.5 Test files

tests/fail-array-append.cqm

```
int main()
{
  int[] arr;
  arr = make(int, 0);

  append(arr, 1.);

  return 0;
}
```

tests/fail-array-concat.cqm

```
int main()
{
  int[] i;
  float[] f;

  i = make(int, 3);
  f = make(float, 3);

  i = concat(i,f);
  return 0;
}
```

tests/fail-assign1.cqm

```
int main()
{
  int i;
  bool b;

  i = 42;
  i = 10;
  b = true;
  b = false;
  i = false; /* Fail: assigning a bool to an integer */
}
```

tests/fail-assign2.cqm

```
int main()
{
  int i;
  bool b;

  b = 48; /* Fail: assigning an integer to a bool */
}
```

tests/fail-assign3.cqm

```
void myvoid()
{
  return;
}

int main()
{
  int i;

  i = myvoid(); /* Fail: assigning a void to an integer */
}
```

tests/fail-dead1.cqm

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

```
4
5     i = 15;
6     return i;
7     i = 32; /* Error: code after a return */
8   }
```

tests/fail-dead2.cqm

```
1   int main()
2   {
3     int i;
4
5     {
6       i = 15;
7       return i;
8     }
9     i = 32; /* Error: code after a return */
10  }
```

tests/fail-def-free.cqm

```
1   void free() {
2     return;
3   }
4
5   void main() {
6     return;
7   }
```

tests/fail-def-len.cqm

```
1   void len() {
2     return;
3   }
4
5   void main() {
6     return;
7   }
```

tests/fail-def-size.cqm

```
void size() {
  return;
}

void main() {
  return;
}
```

tests/fail-expr1.cqm

```
int a;
bool b;

void foo(int c, bool d)
{
  int dd;
  bool e;
  a + c;
  c - a;
  a * 3;
  c / 2;
  d + a; /* Error: bool + int */
}

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

tests/fail-expr2.cqm

```
int a;
bool b;

void foo(int c, bool d)
{
  int d;
  bool e;
  b + a; /* Error: bool + int */
}

int main()
{
```

```
  return 0;
}
```

tests/fail-float-illegal-asn.cqm

```
int main() {
  float f;
  f = 1.0 + 2.0;
  f = 1;
}
```

tests/fail-for1.cqm

```
int main()
{
  int i;
  for ( ; true ; ) {} /* OK: Forever */

  for (i = 0 ; i < 10 ; i = i + 1) {
    if (i == 3) return 42;
  }

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

  return 0;
}
```

tests/fail-for2.cqm

```
int main()
{
  int i;

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

  return 0;
}
```

tests/fail-for3.cqm

```
int main()
{
  int i;

  for (i = 0; i ; i = i + 1) {} /* i is an integer, not Boolean */

  return 0;
}
```

tests/fail-for4.cqm

```
int main()
{
  int i;

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

  return 0;
}
```

tests/fail-for5.cqm

```
int main()
{
  int i;

  for (i = 0; i < 10 ; i = i + 1) {
    foo(); /* Error: no function foo */
  }

  return 0;
}
```

tests/fail-fptr.cqm

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

void print_bin(fp (float, int, int) f, int x, int y) {
  print(f(x, y));
```

```
  return;
}

int main() {
  print_bin(add, 7, 35);

  return 0;
}
```

tests/fail-free.cqm

```
int main()
{
  free(1);
  return;
}
```

tests/fail-free_arr.cqm

```
int main()
{
  free_arr(1);
  return 0;
}
```

tests/fail-func1.cqm

```
int foo() {}

int bar() {}

int baz() {}

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

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

tests/fail-func2.cqm

```
int foo(int a, bool b, int c) { }

void bar(int a, bool b, int a) {} /* Error: duplicate formal a in bar */

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

tests/fail-func3.cqm

```
int foo(int a, bool b, int c) { }

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

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

tests/fail-func5.cqm

```
int foo() {}

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

  return 0;
}

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

tests/fail-func6.cqm

```
void foo(int a, bool b)
{
}

int main()
{
  foo(42, true);
  foo(42); /* Wrong number of arguments */
}
```

tests/fail-func7.cqm

```
void foo(int a, bool b)
{
}

int main()
{
  foo(42, true);
  foo(42, true, false); /* Wrong number of arguments */
}
```

tests/fail-func8.cqm

```
void foo(int a, bool b)
{
}

void bar()
{
}

int main()
{
  foo(42, true);
  foo(42, bar()); /* int and void, not int and bool */
}
```

tests/fail-func9.cqm

```
void foo(int a, bool b)
{
}

int main()
{
  foo(42, true);
  foo(42, 42); /* Fail: int, not bool */
}
```

tests/fail-global1.cqm

```
int c;
bool b;
void a; /* global variables should not be void */


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

tests/fail-global2.cqm

```
int b;
bool c;
int a;
int b; /* Duplicate global variable */

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

tests/fail-if1.cqm

```
int main()
{
  if (true) {}
  if (false) {} else {}
```

```
  if (42) {} /* Error: non-bool predicate */
}
```

tests/fail-if2.cqm

```
int main()
{
  if (true) {
    foo; /* Error: undeclared variable */
  }
}
```

tests/fail-if3.cqm

```
int main()
{
  if (true) {
    42;
  } else {
    bar; /* Error: undeclared variable */
  }
}
```

tests/fail-make-array.cqm

```
int main()
{
  float[] arr;
  arr = make(float[], "what");
  return 0;
}
```

tests/fail-matrix-assign.cqm

```
int main()
{
  fmatrix fm;
  int i;

```

```
  fm = init_fmat_const(2.45, 3, 5);

  fm[1,1] = "what";

  return 0;
}
```

tests/fail-matrix-index.cqm

```
int main()
{
  fmatrix fm;
  int i;

  fm = init_fmat_const(2.45, 1, 6);

  print_float(fm[1.2,1.3]);

  return 0;
}
```

tests/fail-matrix-literal.cqm

```
int main() {
  fmatrix fm;
  fm = [[1.0, 2.0, 3.0], [4.0, 5.0]];
  return 0;
}
```

tests/fail-pipe.cqm

```
int foo(float f)
{
  return 1;
}

int main()
{
  int i;

  i = 0;
```

```
  i => foo();

  return 0;
}
```

tests/fail-return1.cqm

```
int main()
{
  return true; /* Should return int */
}
```

tests/fail-return2.cqm

```
void foo()
{
  if (true) return 42; /* Should return void */
  else return;
}

int main()
{
  return 42;
}
```

tests/fail-struct-assign.cqm

```
struct foo {
    float f;
}

int main()
{
  struct foo foo;
  foo.f = "hello";
}
```

tests/fail-struct-empty.cqm

```
struct foo {
}

int main()
{
  struct foo foo;
}
```

tests/fail-struct-method-dispatch.cqm

```
struct foo {
  float f;
}

struct foo2 {
  float f;
}

[struct foo s] bar() void {
  s.f = 3.14;
  return;
}

int main()
{
  struct foo2 f;

  f = make(struct foo2);
  f.bar();

  return 0;
}
```

tests/fail-while1.cqm

```
int main()
{
  int i;

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

```

```
  while (42) { /* Should be boolean */
    i = i + 1;
  }

}
```

tests/fail-while2.cqm

```
int main()
{
  int i;

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

  while (true) {
    foo(); /* foo undefined */
  }

}
```

tests/test-add1.cqm

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

int main()
{
  print( add(17, 25) );
  return 0;
}
```

tests/test-arith1.cqm

```
int main()
{
  print(39 + 3);
  return 0;
}
```

tests/test-arith2.cqm

```
int main()
{
  print(1 + 2 * 3 + 4);
  return 0;
}
```

tests/test-arith3.cqm

```
int foo(int a)
{
  return a;
}

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

tests/test-arr-to-mat.cqm

```
int main(){
  fmatrix fm;
  int i;
  float[] fa;
  fa = make(float, 20);

  for (i = 0; i < 20; i = i + 1){
    fa[i] = float_of_int(i);
  }
  fm = arr_to_fmat(fa, 4, 5);
  print_mat(fm);

  fm = arr_to_fmat((float[]) {0.0, 1.0, 2.0, 3.0, 4.0, 2.3, 2.4, 2.5, 2.6}, 3,
   ↪  3);
  print_mat(fm);
```

```
15
16    return 0;
17  }
```

tests/test-array-append-struct.cqm

```
1   struct foo {
2     int i;
3   }
4
5   int main()
6   {
7     struct foo foo;
8     struct foo foo2;
9     struct foo[] arr;
10    arr = make(struct foo, 0);
11    print(len(arr));
12    foo = make(struct foo);
13    foo.i = 10;
14    print(foo.i);
15    arr = append(arr, foo);
16    print(len(arr));
17    print(foo.i);
18    foo.i = 9;
19    foo2 = arr[0];
20    print(foo2.i);
21
22    return 0;
23  }
```

tests/test-array-append.cqm

```
1   int main()
2   {
3     float[] f_arr;
4     f_arr = make(float, 0);
5     print(len(f_arr));
6     f_arr = append(f_arr, 1.);
7     print(len(f_arr));
8     print_float(f_arr[0]);
9     f_arr = append(f_arr, 2.4);
10    print(len(f_arr));
11    print_float(f_arr[0]);
```

```
  print_float(f_arr[1]);
  f_arr = append(f_arr, f_arr[0]);
  print_float(f_arr[2]);
  return 0;
}
```

tests/test-array-assign-index.cqm

```
void foo(float[] arr, float f)
{
  arr[2] = f;
}

string[] global_arr;

int main()
{
  float[] arr;
  float[] arr1;
  global_arr = make(string, 1);
  global_arr[0] = "arrays work!";
  print_string(global_arr[0]);
  arr = make(float, 4);
  arr[0] = 1.;
  print_float(arr[0]);
  foo(arr, 2.);
  print_float(arr[2]);
  arr1 = arr;
  print_float(arr1[0]);
  print_float(arr1[2]);
  return 0;
}
```

tests/test-array-assign-struct.cqm

```
struct foo {
  int i;
  float f;
}


int main()
{
```

```
  struct foo foo;
  struct foo foo2;
  struct foo[] foo_arr;
  foo_arr = make(struct foo, 1);
  foo = make(struct foo);
  foo.i = 1;
  foo.f = 3.14;
  print(foo.i);
  print_float(foo.f);

  foo_arr[0] = foo;
  print(foo.i);
  print_float(foo.f);
  foo.i = 2;
  foo.f = 4.12;
  foo2 = foo_arr[0];
  print(foo2.i);
  print_float(foo2.f);

  return 0;
}
```

tests/test-array-concat.cqm

```
int main()
{
  int[] a;
  int[] b;
  int i;

  a = (int[]) {1,2,3};
  b = (int[]) {4,5,6};

  a = concat(a,b);
  print(len(a));

  for (i = 0; i < len(a); i = i + 1) {
    print(a[i]);
  }

  for (i = 0; i < len(b); i = i + 1) {
    print(b[i]);
  }

  return 0;
```

```
}
```

tests/test-array-len.cqm

```
int main()
{
  int[] arr;
  int[] arr2;
  arr = make(int, 19);
  print(len(arr));
  arr2 = arr;
  print(len(arr2));
  return 0;
}
```

tests/test-array-lit.cqm

```
int main()
{
  int i;
  int[] a;
  float[] f;
  a = (int[]) {1,2,3,4};
  f = (float[]) {1.0,2.0,3.0,4.0};
  print(len(a));
  print(len(f));
  for (i = 0; i < len(a); i = i + 1) {
    print(a[i]);
    print_float(f[i]);
  }

  return 0;
}
```

tests/test-array-matrix.cqm

```
int main()
{
  fmatrix[] arr;
  fmatrix fm;
  arr = make(fmatrix, 1);
```

```

  arr[0] = init_fmat_const(2., 2, 3);
  print_mat(arr[0]);

  fm = init_fmat_const(3.14, 4, 5);
  print_mat(fm);

  print_string("\n");

  arr = append(arr, fm);
  print_mat(fm);
  print_string("\n");
  print_mat(arr[0]);
  print_string("\n");
  print_mat(arr[1]);
  print_line();

  fm = fm + 1.;
  print_mat(fm);
  print_line();
  print_mat(arr[1]);
  print_mat(arr[0]);

  return 0;
}
```

tests/test-array-struct.cqm

```
struct foo {
  float f;
  int i;
}

void bar(struct foo foo)
{
  foo.f = 4.5;
}

int main()
{
  struct foo[] arr;
  struct foo foo;
  arr = make(struct foo, 2);
  foo = arr[0];
  bar(foo);
```

```
  foo.i = 2;
  print_float(foo.f);
  print(foo.i);
  return 0;
}
```

tests/test-array-zero-len.cqm

```
int main()
{
  float[] arr;
  arr = make(float, 0);
  print(len(arr));
  return 0;
}
```

tests/test-fib.cqm

```
int fib(int x)
{
  if (x < 2) return 1;
  return fib(x-1) + fib(x-2);
}

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

tests/test-float-comp.cqm

```
int main() {
  float f;
  f = 1.0 + 2.0;
  if (f < 1.5)
```

```
    print(3);
  else
    print(0);
}
```

tests/test-for1.cqm

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

tests/test-for2.cqm

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

tests/test-free.cqm

```
struct foo {
  float f;
}

int main()
{
  struct foo foo;
  foo = make(struct foo);
```

```
  foo.f = 1.;
  free(foo);
  return 0;
}
```

tests/test-free_arr.cqm

```
int main()
{
  int[] a;
  a = make(int, 5);
  print(len(a));
  free_arr(a);
  return 0;
}
```

tests/test-func-pntr.cqm

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

void print_bin(fp (int, int, int) f, int x, int y) {
  print(f(x, y));
  return;
}

int main() {
  print_bin(add, 7, 35);

  return 0;
}
```

tests/test-func-pntr2.cqm

```
void add(int x, int y) {
  print(x+y);
  return;
}

void print_bin(fp (int, int, void) f, int x, int y) {
```

```
  f(x, y);
  return;
}

int main() {
  print_bin(add, 7, 35);

  return 0;
}
```

tests/test-func1.cqm

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

int main()
{
  int a;
  a = add(39, 3);
  print(a);
  return 0;
}
```

tests/test-func2.cqm

```
/* Bug noticed by Pin-Chin Huang */

int fun(int x, int y)
{
  return 0;
}

int main()
{
  int i;
  i = 1;

  fun(i = 2, i = i+1);

  print(i);
  return 0;
}
```

tests/test-func3.cqm

```
void printem(int a, int b, int c, int d)
{
  print(a);
  print(b);
  print(c);
  print(d);
}

int main()
{
  printem(42,17,192,8);
  return 0;
}
```

tests/test-func4.cqm

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

int main()
{
  int d;
  d = add(52, 10);
  print(d);
  return 0;
}
```

tests/test-func5.cqm

```
int foo(int a)
{
  return a;
}

int main()
```

```
{
  return 0;
}
```

tests/test-func6.cqm

```
void foo() {}

int bar(int a, bool b, int c) { return a + c; }

int main()
{
  print(bar(17, false, 25));
  return 0;
}
```

tests/test-func7.cqm

```
int a;

void foo(int c)
{
  a = c + 42;
}

int main()
{
  foo(73);
  print(a);
  return 0;
}
```

tests/test-func8.cqm

```
void foo(int a)
{
  print(a + 3);
}

int main()
{
```

```
  foo(40);
  return 0;
}
```

tests/test-gcd.cqm

```
int gcd(int a, int b) {
  while (a != b) {
    if (a > b) a = a - b;
    else b = b - a;
  }
  return a;
}

int main()
{
  print(gcd(2,14));
  print(gcd(3,15));
  print(gcd(99,121));
  return 0;
}
```

tests/test-gcd2.cqm

```
int gcd(int a, int b) {
  while (a != b)
    if (a > b) a = a - b;
    else b = b - a;
  return a;
}

int main()
{
  print(gcd(14,21));
  print(gcd(8,36));
  print(gcd(99,121));
  return 0;
}
```

tests/test-global1.cqm

```
int a;
int b;

void printa()
{
  print(a);
}

void incab()
{
  a = a + 1;
  b = b + 1;
}

int main()
{
  a = 42;
  b = 21;
  printa();
  incab();
  printa();
  return 0;
}
```

tests/test-global2.cqm

```
bool i;

int main()
{
  int i; /* Should hide the global i */

  i = 42;
  print(i + i);
  return 0;
}
```

tests/test-global3.cqm

```
int i;
bool b;
int j;

```

```
int main()
{
  i = 42;
  j = 10;
  print(i + j);
  return 0;
}
```

tests/test-hello.cqm

```
int main()
{
  print(42);
  print(71);
  print(1);
  return 0;
}
```

tests/test-if1.cqm

```
int main()
{
  if (true) print(42);
  print(17);
  return 0;
}
```

tests/test-if2.cqm

```
int main()
{
  if (true) print(42); else print(8);
  print(17);
  return 0;
}
```

tests/test-if3.cqm

```
int main()
{
  if (false) print(42);
  print(17);
  return 0;
}
```

tests/test-if4.cqm

```
int main()
{
  if (false) print(42); else print(8);
  print(17);
  return 0;
}
```

tests/test-if5.cqm

```
int cond(bool b)
{
  int x;
  if (b)
    x = 42;
  else
    x = 17;
  return x;
}

int main()
{
 print(cond(true));
 print(cond(false));
 return 0;
}
```

tests/test-inline-comment.cqm

```
int main()
{
  print(1);
  // print(2);
```

```
  return 0;
}
```

tests/test-int-float-cast.cqm

```
int main()
{
  int i;
  float f;

  i = 1;
  f = float_of_int(i);
  print_float(f);
  i = int_of_float(f);
  print(i);

  return 0;
}
```

tests/test-link-print.cqm

```
int main() {
  printf("hello world\r\n");
  printf("%f I am a float %d\r\n", 3.14, 29);
  printb(true);
  print(56);
}
```

tests/test-local1.cqm

```
void foo(bool i)
{
  int i; /* Should hide the formal i */

  i = 42;
  print(i + i);
}

int main()
{
  foo(true);
```

```
  return 0;
}
```

tests/test-local2.cqm

```
int foo(int a, bool b)
{
  int c;
  bool d;

  c = a;

  return c + 10;
}

int main() {
 print(foo(37, false));
 return 0;
}
```

tests/test-matdel.cqm

```
int main()
{
  fmatrix[] arr;
  fmatrix fm;

  arr = make(fmatrix, 1);
  arr[0] = init_fmat_identity(4,4);

  print_mat(arr[0]);
  fm = copy(arr[0]);

  fm[1,1] = 1.456;
  print_mat(fm);
  print_mat(arr[0]);
  free_fmat_arr(arr);

  return 0;
}
```

tests/test-mathlib.cqm

```
int main()
{
  float f;
  int i;

  f = 2.0;

  print_float(pow(f,2.));
  print_float(sin(f));
  print_float(cos(f));
  print_float(tan(f));
  print_float(sinh(f));
  print_float(cosh(f));
  print_float(tanh(f));
  print_float(asin(f));
  print_float(acos(f));
  print_float(atan(f));

  print_float(fabs(f));
  print_float(exp(f));
  print_float(log(f));
  print_float(log10(f));
  print_float(sqrt(f));

  /*
  f = 0.;
  for (i = 0; i < 1000000; i = i + 1) {
    f = f + rand_norm(0.0, 1.0);
  }
  print_float(f);
  printf("avg: %f\n", f / 1000000.);
  */

  return 0;
}
```

tests/test-matrix-assign.cqm

```
int main()
{
  fmatrix fm;
  int i;
  int j;

  fm = init_fmat_const(2.45, 3, 5);
```

```
  fm[1,1] = 1.34;
  for (i = 0; i < rows(fm); i = i + 1) {
    for (j = 0; j < cols(fm); j = j + 1) {
      print_float(fm[i,j]);
      if (j == cols(fm) - 1) {
        print_line();
      }
    }
  }

  print_mat(fm);

  return 0;
}
```

tests/test-matrix-index.cqm

```
int main()
{
  fmatrix fm;
  int i;

  fm = init_fmat_const(2.45, 1, 6);

  for (i = 0; i < cols(fm); i = i + 1)
    print_float(fm[0,i]);

  return 0;
}
```

tests/test-matrix-map.cqm

```
float foo(float f){
  return f * 2.0 + 4.0;
}

int main(){
  fmatrix fm;
  fmatrix fm2;
  fmatrix fm3;
  fm = init_fmat_const(4.0, 3, 3);
  fm2 = map(fm, foo);
```

```
  fm3 = map(fm, sqrt);

  print_mat(fm2);
  print_mat(fm3);
  return 0;
}
```

tests/test-matrix-row-col.cqm

```
int main()
{
  fmatrix fm;

  fm = init_fmat_zero(4,5);
  print(rows(fm));
  print(cols(fm));
  fm = init_fmat_zero(3,7);
  print(rows(fm));
  print(cols(fm));

  return 0;
}
```

tests/test-matrix1.cqm

```
extern void onion_matrix_test();

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

tests/test-matrix2.cqm

```
int main(){
        fmatrix fm1;
        fmatrix fm2;
        fmatrix fm3;
```

```
        fm1 = init_fmat_zero(5, 5);
        fm2 = init_fmat_const(2.5, 5, 5);
        fm3 = init_fmat_const(1.23, 2, 8);

        print_mat((fm1 + 1.0) + fm2);
        fm1 = fm1 + 1.0;
        print_mat((fm1 + 12.0) .. fm2);  // matrix mult
        print_mat(fm1 * fm2);         // hadamard product

        print_mat((fm3 + 3.)^);
        print_mat(fm3 + 3.);
        print_line();
        print_mat(3. + fm3);
        print_line();
        print_mat(fm3 - 1.);
        print_line();
        print_string("0. - fm3");
        print_mat(0. - fm3);
        print_line();
        print_string("fm3 / 1.");
        print_mat(fm3 / 1.);
        print_line();
        print_string("1. / fm3");
        print_mat(1. / fm3);
        print_line();
        print_string("fm3 / 3.");
        print_mat(fm3 / 3.);


        return 0;
}
```

tests/test-matrix3.cqm

```
int main(){
  fmatrix fm;
  fmatrix fm2;
  // print_mat([1.0, 2.0, 3.0; 2.0, 3.0, 4.0]);
  fm = [[1.0, 2.0, 3.0], [2.0, 3.0, 4.0]];
  fm2 = [[3.0, 5.0, 6.0], [8.0, 2.3, 5.0], [1.2, 1.3, 1.4]];
  print_mat(fm);
  print_mat(fm2);
  print_mat(fm + 2.0);
  print_mat(fm .. fm2);
  // print_mat(fm);
```

```
  return 0;
}
```

tests/test-multi-decl.cqm

```
int main()
{
  int: i, j;
  int k;
  i = 1;
  j = 2;
  k = 3;
  print(i);
  print(j);
  print(k);

  return 0;
}
```

tests/test-ops1.cqm

```
int main()
{
  print(1 + 2);
  print(1 - 2);
  print(1 * 2);
  print(100 / 2);
  print(99);
  printb(1 == 2);
  printb(1 == 1);
  print(99);
  printb(1 != 2);
  printb(1 != 1);
  print(99);
  printb(1 < 2);
  printb(2 < 1);
  print(99);
  printb(1 <= 2);
  printb(1 <= 1);
  printb(2 <= 1);
  print(99);
  printb(1 > 2);
  printb(2 > 1);
```

```
  print(99);
  printb(1 >= 2);
  printb(1 >= 1);
  printb(2 >= 1);
  return 0;
}
```

tests/test-ops2.cqm

```
int main()
{
  printb(true);
  printb(false);
  printb(true && true);
  printb(true && false);
  printb(false && true);
  printb(false && false);
  printb(true || true);
  printb(true || false);
  printb(false || true);
  printb(false || false);
  printb(!false);
  printb(!true);
  print(-10);
  print(--42);
}
```

tests/test-pipe.cqm

```
struct structer {
  int i;
}

struct structer foo(struct structer a)
{
  print(a.i);
  a.i = a.i + 1;
  return a;
}

struct structer foo1(struct structer a)
{
  print(a.i);
```

```
  a.i = a.i + 1;
  return a;
}

int main()
{
  struct structer s;

  s = make(struct structer);
  s.i = 1;
  s => foo() => foo1();
  print(s.i);

  return 0;
}
```

tests/test-print-float.cqm

```
int main() {
  print_float( 1.0 );
}
```

tests/test-print-string.cqm

```
int main() {
 print_string("hello world");
}
```

tests/test-printbig.cqm

```
/*
 * Test for linking external C functions to LLVM-generated code
 *
 * printbig is defined as an external function, much like printf
 * The C compiler generates printbig.o
 * The LLVM compiler, llc, translates the .ll to an assembly .s file
 * The C compiler assembles the .s file and links the .o file to generate
 * an executable
 */

extern void printbig(int c);
```

```

int main()
{
  printbig(72); /* H */
  printbig(69); /* E */
  printbig(76); /* L */
  printbig(76); /* L */
  printbig(79); /* O */
  printbig(32); /*   */
  printbig(87); /* W */
  printbig(79); /* O */
  printbig(82); /* R */
  printbig(76); /* L */
  printbig(68); /* D */
  return 0;
}
```

tests/test-rec1.cqm

```
void rec(int a) {
    if (a > 5) {
        return;
    }
    print(a);
    rec(a+1);
}

int main() {
    rec(0);
}
```

tests/test-sieve.cqm

```
void sieve_of_eratosthenes(int n)
{
  bool[] prime;
  int: i, p;

  prime = make(bool, n);
  for (i = 0; i < len(prime) + 1; i = i + 1) {
    prime[i] = true;
  }

```

```
  p = 2;
  while (p * p <= n) {
    if (prime[p]) {
      for (i = 2*p; i < len(prime) + 1; i = i + p) {
        prime[i] = false;
      }
    }
    p = p + 1;
  }

  for (i = 2; i < n + 1; i = i + 1) {
    if (prime[i]) {
      print(i);
    }
  }
}

int main()
{
  int n;
  n = 100;

  sieve_of_eratosthenes(n);
}
```

tests/test-struct-array-access.cqm

```
struct foo {
  fmatrix[] fms;
  int[] a;
}

int main()
{
  struct foo foo;
  fmatrix fm;

  foo = make(struct foo);
  foo.a = make(int,5);
  foo.fms = make(fmatrix, 1);
  fm = init_fmat_identity(3,3);
  print_mat(fm);
  foo.fms[0] = init_fmat_identity(4,4);
  print_mat(foo.fms[0]);
  foo.fms[0] = fm;
```

```
  print_mat(foo.fms[0]);
  fm[0,0] = 3.14;
  print_mat(foo.fms[0]);

  return 0;
}
```

tests/test-struct-assign.cqm

```
struct foo {
    float f;
    int i;
}

struct foo foo_global;

int main()
{
  float f;
  struct foo foo;

  foo_global = make(struct foo);
  foo = make(struct foo);

  foo_global.f = 2.4;
  print_float(foo_global.f);

  foo.f = 1.0;
  f = foo.f;
  print_float(f);
  foo.f = 2.0;
  print_float(foo.f);
}
```

tests/test-struct-decl.cqm

```
struct foo {
    float f;
    int i;
}

void struct_function(struct foo f)
{
```

```
  return;
}

int main()
{
  float what;
  struct foo f1;
  struct foo f2;

  f1 = make(struct foo);
  f2 = f1;
}
```

tests/test-struct-func.cqm

```
struct foo {
    float f;
    int i;
}

void struct_function(struct foo f, int i)
{
  f.i = i;
  return;
}

int main()
{
  float what;
  struct foo f1;

  f1 = make(struct foo);
  struct_function(f1, 2);
  print(f1.i);
  struct_function(f1, 3);
  print(f1.i);
}
```

tests/test-struct-matrix.cqm

```
struct foo {
  fmatrix fm;
}
```

```

int main()
{
  struct foo foo;
  foo = make(struct foo);

  foo.fm = init_fmat_const(2., 3, 3);
  print_mat(foo.fm);

  return 0;
}
```

tests/test-struct-method-dispatch.cqm

```
struct foo {
  float f;
}

[struct foo s] bar() void {
  s.f = 3.14;
  return;
}

int main()
{
  struct foo f;

  f = make(struct foo);
  f.bar();
  print_float(f.f);

  return 0;
}
```

tests/test-struct-nested.cqm

```
struct inner_foo {
  float f;
}

struct foo {
  int i;
  struct inner_foo inner_foo;
```

```
}

int main()
{
  struct foo foo;
  struct inner_foo inner_foo;
  struct inner_foo inner_foo2;

  foo = make(struct foo);
  inner_foo = make(struct inner_foo);

  inner_foo.f = 3.14;
  print_float(inner_foo.f);

  foo.inner_foo = inner_foo;
  inner_foo2 = foo.inner_foo;

  print_float(inner_foo2.f);

  inner_foo2.f = 1.23;
  print_float(inner_foo.f);

  return 0;
}
```

tests/test-struct-of-array.cqm

```
struct foo {
  int[] a;
}

int main()
{
  struct foo foo;
  int[] a;
  foo = make(struct foo);
  foo.a = make(int, 5);
  a = foo.a;
  a[0] = 1;
  print(a[0]);
  print(len(a));
  return 0;
}
```

tests/test-var1.cqm

```
int main()
{
  int a;
  a = 42;
  print(a);
  return 0;
}
```

tests/test-var2.cqm

```
int a;

void foo(int c)
{
  a = c + 42;
}

int main()
{
  foo(73);
  print(a);
  return 0;
}
```

tests/test-while1.cqm

```
int main()
{
  int i;
  i = 5;
  while (i > 0) {
    print(i);
    i = i - 1;
  }
  print(42);
  return 0;
}
```

tests/test-while2.cqm

```
int foo(int a)
{
  int j;
  j = 0;
  while (a > 0) {
    j = j + 2;
    a = a - 1;
  }
  return j;
}

int main()
{
  print(foo(7));
  return 0;
}
```