BLAStoff Report

Katon Luaces, Michael Jan, Jake Fisher, Jason Kao
{knl2119, mj2886, jf3148, jk4248}@columbia.edu

Contents

1 Introduction

Expressing an algorithm primarily through manipulation of matrices allows an implementation to take advantage of parallel computation. Graphs are one of the most important abstract data structures and graph algorithms underlie a wide range of applications. Yet many implementations of graph algorithms rely on sequential pointer manipulations that cannot easily be parallelized. As a result of the practicality and theoretical implications of more efficient expressions of these algorithms, there is a robust field within applied mathematics focused on expressing "graph algorithms in the language of linear algebra"[KG11]. BLAStoff is a linear algebraic language focused on the primitives that allow for efficient and elegant expression of graph algorithms.

2 Tutorial

2.1 Language Details

2.1.1 Data Types

There is only one data type in BLAStoff, the matrix. Matrices can be defined four ways: as a matrix literal, as a graph, as a number, or with a generator function.

2.1.1.1 Matrix Literal Definition

A matrix literal looks as follows:

```
M = [1,3,5;
     2,4,6;
     0,0,-1];
```

which sets M as the matrix

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \\ 0 & 0 & -1 \end{bmatrix}$$

. The values given to M can be anything $\in \mathbb{R} \cup \pm\infty$. Here's an example of using values that aren't just integers:

```
M = [1.2, inf;
    -inf, -34];
```

which sets M as the matrix

$$\begin{bmatrix} 1.2 & \infty \\ -\infty & -34 \end{bmatrix}$$

.

2.1.1.2 Graph

The graph definition looks as follows:

```
G = {
    0->1;
    1->0;
    1->2;
    4;
};
```

This will set M as the adjacency matrix for the graph described, which in this case would be:

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

As we can see in this code example, each line in the graph definition can be an edge $a \to b$; defining a node between vertices a and b where a, b are non-negative integers, or just a vertex c; where c is also a non-negative integer, which just defines that the vertex c exists. The matrix created will be an $n \times n$ matrix, where n is the highest vertex (in our case 4) defined plus 1. Thus, the graph created will have nodes $[0, n-1]$. Any vertices not mentioned in the definition but in the range $[0, n-1]$ will be created, but not have any edges to or from it (such as vertex 3 in this case). We could easily extend this syntax to allow for bi-directional or weighted graphs, and will decide later whether to do this.

2.1.1.3 Number Definition

The number definition is quite simple, and looks like as follows:

```
M = 5;
```

This is how you would create a "scalar" in BLAStoff, but because the only data type is a matrix, scalars are really 1×1 matrices. The above code is equivalent to the following code:

```
M = [5];
```

which sets M as the matrix

$$\begin{bmatrix} 5 \end{bmatrix}$$

A potential issue could be that not allowing scalars means you can't have scalar multiplication, but we define a convolution operator, which ends up working like scalar multiplication if the right hand side is a 1×1 matrix.

2.1.1.4 Generator Function Definition

We also have a number of generator functions for commonly-used types of matrices so that you don't waste your time typing out a 50×50 identity matrix. The first is the `Zero` function, which generates a matrix with all 0s. This takes in one argument, which we will call x, a non-negative integer matrix of two possible sizes. n can be a 2×1 positive integer matrix, and the elements of the n matrix are the height and width of the zero matrix, in that order. n could also be a 1×1 matrix, in which case the zero matrix will be square, with the element in n as its height and width. Here is an example:

```
A = Zero(4);
B = Zero([3;2]);
```

This code would result in the following matrices:

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Note that `A = Zero(4);` is equivalent to `A = Zero([4;4]);`.

We also have an identity function, `I`, which takes in one argument, a 1×1 non-negative integer matrix, the width and height of the resultant square identity matrix. Example:

```
M = I(3);
```

This would result in the following matrix:

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The final generator function is the `range` function, which generates a column vector that goes through an integer range, incremented by 1. Like `Zero`, it takes in an integer matrix of size 1×1 or size 2×1, which gives the bounds of the range generated (inclusive lower, exclusive upper), or, in the 1×1 case, the exclusive upper bound, and 0 is the default lower bound. Here are some examples:

```
A = range(3);
B = range(-2,2);
```

This code would result in the following matrices:

$$A = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \end{bmatrix}$$

$$B = \begin{bmatrix} -2 \\ -1 \\ 0 \\ 1 \\ 2 \end{bmatrix}$$

If a range where the lower bound is greater than the upper bound given to `range`, such as `range([5;-1])`, a 0×1 matrix will be returned.

2.1.1.5 Integers vs. Floats

You're probably confused now, because I said earlier that the only type in BLAStoff is a matrix, but now I'm talking about integers and floats? So, while in a perfect world we could just have everything be floats, as we're defining our linear algebra over the reals, consider the following code if (which makes use of a few operators we will define below, but you can guess how it generally works for now):

```
b = 25020359023950923059124;
a = 2;
M = [1,2;3,4];
a += b;
a -= b;
M = M^a;
```

If this code has no floating point errors, than the final line is just a simple matrix squaring. However, if some error is introduced to `a`, then we have a problem where we're trying to calculate something like $M^{2.000001}$, which is a much more difficult problem even if it would result in a numerically similar result. So, I was lying a little. Though you don't declare any types explicitly, each matrix is implicitly a float matrix or an integer matrix depending on if it is defined with any non-integers (you can only get float matrices with the literal definition). Any operation (such as matrix addition or matrix multiplication) between a float matrix and an integer matrix results in a float matrix, while an operation between two matrices of the same type will result in a matrix of the same type, except for something like matrix inversion.

2.1.2 Comments

Comments in BLAStoff use C style, with `//` for a single line and `/*` and `*/` for multi-line comments. For example:

```
A = 6; // I'm a comment!
B = 5; /* I'm a comment also but
...
...
I'm longer!*/
```

2.1.3 Functions

Functions in BLAStoff are defined with a mix of Python and C style:

```
def foo(A, B) {
    return A;
}
```

Because there is only one data type in BLAStoff, there is no need for argument types or return types, everything is always a matrix! Even "void" functions return matrices. Consider these two functions:

```
def bar1(A) {
    return;
}

def bar2(A) {
    ;
}
```

These two functions both return the equivalent of Python's "None" in BLAStoff, a 0×0 matrix.

2.1.4 If statements

For and while loops also look similar to C. For example:

```
if (A > 2) {
    A = 7;
} else if (A < -3) {
    A = 5;
} else {
    A = 0;
}
```

2.1.5 For/While Loops

For and while loops also look similar to C. For example:

```
B = 0;
for (A = [0]; A < 5 ; A+=1) {
```

```
    B+=1;
}

while (B > -1) {
    B-=1;
}
```

We allow for loops, but they are not usually the ideal paradigm. The selection operator should hopefully replace much of the use for loops.

2.1.6 Operations

Operations are where BLAStoff gets more interesting.

We aim to implement a large subset of the basic primitives described in [Gil] (several of which can be combined) as well as a few essential semirings.

Semiring	operators $\oplus$	operators $\otimes$	domain	**0**	**1**
Standard arithmetic	$+$	$\times$	$\mathbb{R}$	0	1
max-plus algebras	max	$+$	$\{-\infty \cup \mathbb{R}\}$	$-\infty$	0
min-max algebras	min	max	$\infty \cup \mathbb{R}_{\geq 0}$	∞	0
Galois fields (*e.g.*, GF2)	xor	and	$\{0, 1\}$	0	1
Power set algebras	$\cup$	$\cap$	$\mathcal{P}(\mathbb{Z})$	$\varnothing$	U

Semirings.png

Operation name	Mathematical description
mxm	$\mathbf{C} \odot= \mathbf{A} \oplus . \otimes \mathbf{B}$
mxv	$\mathbf{w} \odot= \mathbf{A} \oplus . \otimes \mathbf{v}$
vxm	$\mathbf{w}^T \odot= \mathbf{v}^T \oplus . \otimes \mathbf{A}$
eWiseMult	$\mathbf{C} \odot= \mathbf{A} \otimes \mathbf{B}$
	$\mathbf{w} \odot= \mathbf{u} \otimes \mathbf{v}$
eWiseAdd	$\mathbf{C} \odot= \mathbf{A} \oplus \mathbf{B}$
	$\mathbf{w} \odot= \mathbf{u} \oplus \mathbf{v}$
reduce (row)	$\mathbf{w} \odot= \bigoplus_j \mathbf{A}(:, j)$
apply	$\mathbf{C} \odot= F_u(\mathbf{A})$
	$\mathbf{w} \odot= F_u(\mathbf{u})$
transpose	$\mathbf{C} \odot= \mathbf{A}^T$
extract	$\mathbf{C} \odot= \mathbf{A}(\mathbf{i}, \mathbf{j})$
	$\mathbf{w} \odot= \mathbf{u}(\mathbf{i})$
assign	$\mathbf{C}(\mathbf{i}, \mathbf{j}) \odot= \mathbf{A}$
	$\mathbf{w}(\mathbf{i}) \odot= \mathbf{u}$

primitives.png

Here are our operations, which implement those and more:

Name	Usage
Assigmnent	M = N
Selection	M[A, B, c, d]
Matrix Multiplication	M * N
Convolution	M ~ N
Element-wise Multiplication	M @ N
Element-wise Addition	M + N
Exponentiation/Inverse/Transpose	A^b or A^T
Size	|M|
Vertical Concatenation	M : N
Reduce Rows	+$M or *$M
Semiring Redefinition	<#semiringname> or <_>
Logical Negation	<#semiringname> or <_>
Comparisons	==, !=, >, >=, <, <=
Assignment Variants	*=, .=, @=, +=, ^=, :=

Some of these operators' behaviors are intuitive, but we will explained further the two less intuitive ones:

2.1.6.1 Selection []

The BLAStoff selection operator can be applied to any matrix looks like one of the following three forms:

```
M[A, B, c, d];
M[A, B]
M[A];
```

where A, B are column vectors of non-negative integers ($n \times 1$ matrices) and c, d are 1×1 non-negative integer matrices. c, d are optional and have a default value of $\begin{bmatrix}1\end{bmatrix}$. B is also optional and its default value is $\begin{bmatrix}0\end{bmatrix}$. Abstractly, the way this operator works is by taking the Cartesian product of A, B, $R = A \times B$, and for each $(j, i) \in R$, we select all the sub-matrices in M with a top-left corner at row j, column i, height of c, and width of d. (BLAStoff is 0-indexed.) This Cartesian makes the select operator a very powerful operator that can do things like change a specific of indices, while also being general enough to allow for simple indexing. Take the following code example:

```
M = Zero(4)
M[[0;2], [0;2]] = 1;
```

This would result in the following matrix:

$$M = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

as in this case $R = \{(0,0),(0,1),(1,0),(1,1)\}$, so for every 1×1 matrix at each point in R, we set the value to 1. Note that the matrix on the right hand side must be of size $c \times d$. That was a relatively complicated use of the select operator, but simple uses still have very easy syntax:

```
M = Zero(2);
M[1, 0] = 1;
N = Zero(3);
N[1, 1, 2, 2] = I(2);
```

This would result in:

$$M = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$$

$$N = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The reason why 0 is the default value of B is to allow for easy column vector access. Example:

```
v = [1;1;1];
v[1] = 2;
u = [1;1;1];
u[[0;2]] = 2;
```

This would result in:

$$v = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$$

$$u = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix}$$

Now, perhaps it is clear why we included the `range` generator function. Example:

```
v = Zeroes([5;1]);
v[range(5)] = 1;
```

This would result in:

$$v = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

As you'd expect, trying to access anything out-of-bounds with the selection operator will throw an error.

We have shown the selection operator so far as a way of setting elements in a matrix, but it's also a way of extracting values from a matrix, as we will show below:

```
A = [1,2,3;
    4,5,6;
    7,8,9];
B = A[0, 0, 2, 2];
```

This would result in:

$$B = \begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}$$

Extraction is quite understandable when A and B are 1×1, as that results in only one matrix, but it is a bit more complicated when they are column vectors. In that case, we concatenate the number of resultant matrices, both vertically and horizontally. I think an example makes this clearer:

```
A = [1,2,3;
    4,5,6;
    7,8,9];
B = A[[0;2], [0;2] , 1, 1];
v = [1;2;3;4];
u = v[[0;2;3]];
```

This would result in:

$$B = \begin{bmatrix} 1 & 3 \\ 7 & 9 \end{bmatrix}$$

$$u = \begin{bmatrix} 1 \\ 3 \\ 4 \end{bmatrix}$$

2.1.6.2 Semiring redefinition <>

You may have noticed that though we have defined a number of operations on matrices, we want a way to control which operations are used on the elements of these matrices beyond standard arithmetic + and ×. We want to be able to use a number of semiring operators, such as those defined in the image above. BLAStoff allows for semiring redefinition in one of the following forms:

```
<#logical>
<#arithmetic>
<#maxmin>
<_>
```

So what does this syntax actually do? Ignore the underscore case for now. The other three are commands to switch the command to the one denoted in the brackets. Let's see an example:

```
a = 2.1;
b = 3;
c = 0;

<#arithmetic>;
a + b; //returns 5.1
a * b; //returns 6.3
a * c; //returns 0

<#logical>;
a + b; //returns 1: plus is now logical or; 0 is the only false value
    and 1 is the default true value
a * b; //returns 1 as well: times is now logical and
a * c; //returns 0


<#maxmin>;
b + c; //returns 2.1; plus is now minimum
a * b; //returns 3; times is now maximum
a * c; //returns 5.1
```

`#arithmetic` is the default, so that line was technically redundant, but included for clarity. The example we gave was with 1×1 matrices, but the semiring definitions work on matrices of any size:

```
A = [1,4;
    6,3];
B = [5,2;
    7,1];
C = A+B;
```

This would result in:

$$C = \begin{bmatrix} 1 & 2 \\ 6 & 1 \end{bmatrix}$$

Semiring redefinition generally is reset back to the default arithmetic when you call a function:

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

a = 4;
b = 3;
<#logical>;

a + b; // will return 1
add(a, b); // will return 7
```

But we provide the `<_>` in order to solve this: calling that command will set the semiring to whatever it was as this function was called (or to arithmetic as a default if you're not in a function):

```
def semiringAdd(x, y) {
    <_>;
    return x + y;
}

a = 4;
b = 3;
<#logical>;

a + b; // will return 1
semiringAdd(a, b); // will also return 1
```

2.1.6.3 Extensions

We can possibly add support for max-plus or Galois, but that will be a stretch goal. Another, possibly loftier stretch goal, is to allow custom semirings:

```
<+, f1, i1>
<*, f2, i2>
```

where f_1, f_2 is the name a function defined elsewhere that has exactly 2 arguments, and i_1, i_2 are number literals. This would use $f_1(a,b)$ for $a+b$ and $f_2(a,b)$ for $a \times b$. i_1, i_2 are the new empty sum and product. We will determine if this is easy/feasible/useful/reasonable.

2.1.7 Keywords

BLAStoff reserves the following keywords:

`I, Zero, range, def, return, if, else, for, while, T`

2.1.8 Memory

BLAStoff will use pass-by-reference and copy-by-value. Here's an example of how this will works:

```
def f(x){
    x += 1;
}
a = 1;
f(a);
a == 1; //TRUE
a == 2; //TRUE

```

```
b = 1;
c = b;
c += 1;
c == 2; //TRUE
b == 2; //FALSE
b == 1; //TRUE
```

BLAStoff will be garbage-collected.

2.1.9 Scope

BLAStoff has scope shared between blocks in the same function call, but not in different function calls. Example:

```

a = 1;
{
    b = 2 + a; // valid
}
c = b + 1; // valid

def f(x){
    return x * (b + c); // compile-time error
}
```

2.1.10 Libraries/Importing Functions

There will be a way to make/use a library and import functions, but we have not settled on the syntax nor semantics.

2.1.11 I/O

There will be a way to use input and output, but we also have not settled on the syntax nor semantics.

2.2 Sample Code

2.2.1 Some Standard Library Functions

As we have discussed, we intend to provide a standard library that should have include a good number of the other Linear Algebra operations that aren't primitives. Here are some examples:

2.2.1.1 One

`One` works exactly like `Zero`, but has all 1s in the matrix:

```
def One(size){
    A = Zero(size);
    m = size[0];
    A[range(size[0]), range(size[1])] = 1;
    return A;
}
```

2.2.1.2 Horizontal Concatenation

We don't include this as an operator because it is quite easy to write as a function using vertical concatenation and transpose:

```
def horizontalConcat(A, B){
    return (A^T:B^T)^T;
}
```

2.2.1.3 Plus/Times Column Reduce

Column reduction follows similarly:

```
def plusColumnReduce(A){
    <_>;
    return ((+$A)^T)^T;
}

def timesColumnReduce(A){
    <_>;
    return ((*$A)^T)^T;
}
```

2.2.1.4 Sum

`sum` gives you the sum of all the elements in the matrix. There are two simple $O(N)$ implementations (where N is the total number of elements in the matrix), and I'll provide both options as an example:

```
def sum(A){
    <_>;
    return A~One(|A|);
}

def sum(A){
    <_>;
    return plusColumnReduce(+$A);
}
```

2.2.1.5 Range From Vector

`rangeFromVector` takes in a column vector and returns a vector of the indices that have non-zero. For instance:

$$\text{rangeFromVector}(\begin{bmatrix}0\\1\\1\\0\\1\end{bmatrix}) = \begin{bmatrix}1\\2\\4\end{bmatrix}$$

This will come in handy in the BFS algorithm that we will write:

```
def rangeFromVector(v){
    <#logical>;
    vlogic = v~1;
    <#arithmetic>;
    n = plusColumnReduce(v); // the number of non-zero values
    u = Zero(n, 1);
    j = 0;
    for (i = 0; i < |v|[0]; i += 1) {
        if (v[i]) {
            u[j] = i;
            j++;
        }
    }
}
```

2.2.2 Graph Algorithms: Breadth First Search

Here we demonstrate how pseudocode from a 2019 presentation by John Gilbert describing BFS in linear algebraic terms [Gil] can be expressed in BLAStoff

BFS pseudocode.png

```
Input: graph, frontier, levels
depth ← 0
while nvals(frontier) > 0:
  depth ← depth + 1
  levels[frontier] ← depth
  frontier<¬levels,replace> ← graph^T ⊕.⊗ frontier
    where ⊕.⊗ = ⨁.⨂(LogicalSemiring)
```

Our correspondong code for BFS looks like the following:

```
def BFS(G, frontier){
    <#logical>;
    N = |G|[0];
    levels = Zero(N, 1);
    maskedGT = G^T;
    depth = 0;
    while (sum(frontier)) {
```

```
8          <#arithmetic>;
9          depth += 1;
10         <#logical>;
11         levels[rangeFromVector(frontier)] = depth;
12         mask = !(frontier^T)[Zeroes(N), 0, 1, N];
13         maskedGT @= mask;
14         frontier = maskedGT*frontier;
15     }
16     <#arithmetic>;
17     return levels + (One(|levels|)~(-1));
18 }
```

Let's look at how this code works. (Note: the cited slides can be helpful for understanding the linear algebra aspects of the algorithm.). It takes in an $n \times n$ adjacency matrix G and a column vector $frontier$ of height n as well, where each entry is 0 or a true value, to denote whether that vertex is in the starting list. On line 4, we then create *levels*, a vector of the same size as $frontier$. This will be our output vector, as it $levels[i]$ will contain the closest distance from vertex i to a vertex in frontiers, or -1 if its unreachable. You'll notice that we initialize *levels* with 0s as we will decrement on line 17. We then make a new variable *maskedGT* on line 5, which is just the transpose of G. We do this because we are going to be modifying this matrix, but we don't want to change the original G. We take the transpose because that's what allows for part of the algorithm, which I'll explain in a second, and we don't want to do that on every iteration. We then set a variable *depth* to 0 on 6. This will keep track of our iterations.

Then we start the while loop, which keeps going as long as there is one non-zero value in $frontier$; that is, we still have vertices we want to look at. We then increment depth on line 9, switching quickly to arithmetic for this one line, as otherwise depth would never go above 1. Using our range-from-vector function defined in the standard library, line 11 essentially sets $levels[i]$ equal to the current depth if $frontier[i]$ is non-zero. That way, all the vertices that we're currently searching for have their distance in levels as the current iteration in our while loop. This will be one more than the level, but we're going to decrement on line 17. The key portion of this code is line 14, which multiplies *maskedGT* and $frontier$. Because of the way the adjacency matrix is constructed, this will give us a vector in the same format as $frontier$, only now with the vertices reachable from the vertices in the original $frontier$, and we will overwrite $frontier$ with this new frontier. With all that I've explained so far, the algorithm would be give you the correct reachable nodes, but would run over paths to vertices for which we've already found a closer path, so depths would be wrong.

To account for this issue, on lines 12 and 13 we remove all the edges to the nodes in frontier, so that as we continue in BFS, we add a previously visited node. We generate a mask by taking our frontier, transposing it, concatenating it down N times, and negating it. Here's an example:

$$frontier = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

In this map, all the ones denote edges not to items in frontier, and thus edges we can keep. So, if we do element-wise multiplication between this mask matrix and our ongoing, masked, G^T, we will keep removing those edges and ensure we never revisit!

3 Project Plan

4 Test Plan

5 Lessons Learned

5.1 Katon

5.2 Michael

5.3 Jake

5.4 Jason

6 Appendix

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

type action =
  | Ast
  | Semant
  | LLVM_IR
  | Compile

let () =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist =
    [ "-a", Arg.Unit (set_action Ast), "Print the AST"
    ; "-s", Arg.Unit (set_action Semant), "Print the SAST"
    ; "-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR"
    ; ( "-c"
      , Arg.Unit (set_action Compile)
      , "Check and print the generated LLVM IR (default)" )
    ]
  in
  let usage_msg = "usage: ./blastoff.native [-a|-s|-l|-c] [file.blst]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;
  let lexbuf = Lexing.from_channel !channel in
  let scanner_token_wrapper lb =
    let tok = Scanner.token lb in
    tok
  in
  let ast = Blastoffparser.program scanner_token_wrapper lexbuf in
  match !action with
  | Ast -> print_string (Ast.string_of_program ast)
  | _ ->
    let sast =
      try Semant.check ast with
      | e ->
        let msg = Printexc.to_string e in
        raise (Failure ("Semantic Checking Error: " ^ msg))
    in
    (match !action with
    | Ast -> ()
    | Semant -> print_string (Ast.string_of_program sast)
    | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate sast))
    | Compile ->
      let m = Codegen.translate sast in
```

```
    Llvm_analysis.assert_valid_module m;
    print_string (Llvm.string_of_llmodule m))
;;
```

```
(* Ocamllex scanner for BLAStoff *)

{ open Blastoffparser

(* http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual026.html#toc111 *)
let keyword_table = Hashtbl.create 97
let _ = List.iter (fun (kwd, tok) -> Hashtbl.add keyword_table kwd tok)
    [ "while", WHILE;
      "return", RETURN;
      "if", IF;
      "else", ELSE;
      "for", FOR;
      "def", FDECL;
      "T", TRANSP]
}

let digit = ['0'-'9']
let arrow = ['-']['>']

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"    { comment lexbuf }         (* Comments *)
| "//"    { single_line_comment lexbuf }
| '-'?digit* as lxm { INTLITERAL(int_of_string lxm) }
| ['-']?digit*['.']digit* as lxm { FLOATLITERAL(float_of_string lxm) }
| '|'     { VLINE }
| '['     { LBRACK }
| ']'     { RBRACK }
| '('     { LPAREN }
| ')'     { RPAREN }
| '{'     { LBRACE }
| '}'     { RBRACE }
| '\''[^'\'']*'\'' as str { STRINGLITERAL(String.sub str 1 ((String.length str) - 2)) }
| '@'     { ELMUL  }
| "@="    { ELMULASSIGN }
| '~'     { CONV }
| "~="    { CONVASSIGN }
| ':'     { CONCAT }
| ":="    { CONCATASSIGN }
| ';'     { SEMI }
| ','     { COMMA }
| '+'     { PLUS }
| "+="    { PLUSASSIGN }
```

```
| '*'     { MATMUL }
| "*="    { MATMULASSIGN }
| '='     { ASSIGN }
| arrow    { EDGE }
| ['+']['%']    { PLUSREDUCE }
| ['*']['%']    { MULREDUCE }
| "=="    { EQ }
| "!="    { NEQ }
| '<'     { LT }
| "<="    { LEQ }
| ">"     { GT }
| ">="    { GEQ }
| '^'     { RAISE }
| "^="    { RAISEASSIGN }
| '!'     { NOT }
| '#'     { SEMIRING }
| ['a'-'z' 'A'-'Z' '_']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm
  { (*print_endline "find lxm: ";
      print_endline lxm;*)
    try
      Hashtbl.find keyword_table lxm
    with Not_found ->
      ID(lxm)}
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
      "*/" { token lexbuf }
  | _    { comment lexbuf }
and single_line_comment = parse
  '\n' { token lexbuf }
  | _ { single_line_comment lexbuf }
```

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

type op =
  | Add
  | Matmul
  | Elmul
  | Conv
  | Equal
  | Neq
  | Less
  | Leq
  | Greater
  | Geq
  | Concat
  | Exponent

```

```
type uop =
  | Neg
  | Transp
  | Plusreduce
  | Mulreduce
  | Size

type lit =
  | IntLit of int
  | FloatLit of float

type expr =
  | GraphLit of (int * int) list
  | UnkMatLit of lit list list
  | IntMatLit of int list list
  | FloatMatLit of float list list
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of expr * expr
  | IdAssign of string * expr
  | SelectAssign of string * expr list * expr
  | Selection of expr * expr list
  | Call of string * expr list
  | StringLit of string

type stmt =
  | Semiring of string
  | Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | While of expr * stmt

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

type program = func_decl list * stmt list

(* Pretty-printing functions *)

let string_of_op = function
  | Add -> "+"
  | Matmul -> "*"
  | Elmul -> "@"
  | Conv -> "~"
  | Equal -> "=="
```

```
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | Exponent -> "^"
  | Concat -> ":"
;;

let string_of_mat lit_to_string m =
  let string_of_row row =
    String.concat "," (List.fold_left (fun acc lit -> lit_to_string lit
        :: acc) [] row)
  in
  "["
  ^ String.concat ";" (List.fold_left (fun acc row -> string_of_row row
      :: acc) [] m)
  ^ "]"
;;

let string_of_graph g =
  let string_of_edge (v1, v2) = string_of_int v1 ^ "->" ^ string_of_int
      v2 in
  "[" ^ String.concat ";" (List.map string_of_edge g) ^ "]"
;;

let rec string_of_expr = function
  | Id s -> s
  | Binop (e1, o, e2) ->
    string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop (o, e) -> string_of_e_with_uop e o
  | Assign (e1, e2) -> string_of_expr e1 ^ " = " ^ string_of_expr e2
  | IdAssign (s, e) -> s ^ " = " ^ string_of_expr e
  | Call (f, el) -> f ^ "(" ^ String.concat ", " (List.map
      string_of_expr el) ^ ")"
  | UnkMatLit m ->
    string_of_mat
      (fun lit ->
        match lit with
        | IntLit ilit -> string_of_int ilit
        | FloatLit flit -> string_of_float flit)
      m
  | IntMatLit m -> string_of_mat string_of_int m
  | GraphLit g -> string_of_graph g
  | StringLit s -> "'" ^ s ^ "'"
  | FloatMatLit m -> string_of_mat string_of_float m
  | Selection (e, args) ->
    string_of_expr e ^ "[" ^ String.concat ", " (List.map string_of_expr
        args) ^ "]"
  | SelectAssign (s, args, e) ->
```

```
    s
    ^ "["
    ^ String.concat ", " (List.map string_of_expr args)
    ^ "]"
    ^ " = "
    ^ string_of_expr e

and string_of_e_with_uop e =
  let str_expr = string_of_expr e in
  function
  | Neg -> "!" ^ str_expr
  | Size -> "|" ^ str_expr ^ "|"
  | Transp -> str_expr ^ "^T"
  | Plusreduce -> "+%" ^ str_expr
  | Mulreduce -> "*%" ^ str_expr
;;

let rec string_of_stmt = function
  | Semiring ring -> "#" ^ ring ^ "\n"
  | Block stmts -> "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr expr -> string_of_expr expr ^ ";\n"
  | Return expr -> "return " ^ string_of_expr expr ^ ";\n"
  | If (e, s, Block []) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If (e, s1, s2) ->
    "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | While (e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
;;

let string_of_func func =
  "def "
  ^ func.fname
  ^ "("
  ^ String.concat ", " func.formals
  ^ ")"
  ^ "{\n"
  ^ String.concat "" (List.map string_of_stmt func.body)
  ^ "}\n"
;;

let string_of_program (funcs, stmts) =
  String.concat "" (List.map string_of_func funcs)
  ^ "\n"
  ^ String.concat "" (List.map string_of_stmt stmts)
;;
```

```
/* Ocamlyacc parser for BLAStoff */

%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK COMMA SEMIRING EDGE
%token MATMUL ELMUL ASSIGN FDECL RANGEMAT CONV PLUS RAISE PLUSREDUCE
    MULREDUCE
%token NOT EQ NEQ LT LEQ GT GEQ IMAT ELMAT TRANSP VLINE SEMIRING CONCAT
    ZEROMAT
%token RETURN IF ELSE FOR WHILE INT BOOL FLOAT VOID
%token PLUSASSIGN ELMULASSIGN CONVASSIGN MATMULASSIGN CONCATASSIGN
    RAISEASSIGN
%token <int> INTLITERAL
%token <float> FLOATLITERAL
%token <string> STRINGLITERAL
%token <string> ID
%token EOF

%start program
%type <Ast.program> program

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN PLUSASSIGN ELMULASSIGN CONVASSIGN MATMULASSIGN
    CONCATASSIGN RAISEASSIGN
%left EQ NEQ
%left LT GT LEQ GEQ
%right LBRACK RBRACK
%left PLUS
%left MATMUL ELMUL
%left CONCAT CONV
%left RAISE
%left EDGE
%right PLUSREDUCE MULREDUCE
%left TRANSP
%right NOT
%%

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

units:
    /* empty */ { ([], []) }
    | units fdecl { ($2 :: fst $1 , snd $1) }
    | units stmt { (fst $1, $2 :: snd $1) }

fdecl:
```

```
   FDECL ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
   { { fname = $2;
       formals = $4;
       body = List.rev $7 } }

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

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

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

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

stmt:
    expr SEMI                                { Expr $1
                                           }
  | SEMIRING ID SEMI                         { Semiring $2
                                     }
  | RETURN ret_opt 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)
                                  }
  | WHILE LPAREN expr RPAREN stmt           { While($3, $5)
                                   }
  | FOR LPAREN stmt expr SEMI expr RPAREN stmt { Block([$3 ; While($4,
      Block([$8 ; Expr($6)]))])}

ret_opt:
      /* nothing */ { UnkMatLit([[]]) }
  | expr        { $1 }


lit:
    INTLITERAL { IntLit($1) }
  | FLOATLITERAL { FloatLit($1) }

expr:
   lit        { UnkMatLit([[$1]]) }
```

```
  | STRINGLITERAL { StringLit($1) }
  | ID             { Id($1)              }
  | expr PLUS  expr { Binop($1, Add, $3)  }
  | ID PLUSASSIGN expr { IdAssign($1, Binop(Id($1), Add, $3)) }
  | expr MATMUL expr { Binop($1, Matmul, $3) }
  | ID MATMULASSIGN expr { IdAssign($1, Binop(Id($1), Matmul, $3)) }
  | expr ELMUL expr { Binop($1, Elmul, $3) }
  | ID ELMULASSIGN expr { IdAssign($1, Binop(Id($1), Elmul, $3)) }
  | expr EQ    expr { Binop($1, Equal, $3) }
  | expr NEQ   expr { Binop($1, Neq, $3)  }
  | expr LT    expr { Binop($1, Less, $3) }
  | expr LEQ   expr { Binop($1, Leq, $3)  }
  | expr GT    expr { Binop($1, Greater, $3) }
  | expr GEQ   expr { Binop($1, Geq, $3)  }
  | expr CONV  expr { Binop($1, Conv, $3) }
  | ID CONVASSIGN expr { IdAssign($1, Binop(Id($1), Conv, $3)) }
  | expr CONCAT  expr { Binop($1, Concat, $3)}
  | ID CONCATASSIGN expr { IdAssign($1, Binop(Id($1), Concat, $3)) }
  | expr RAISE expr { Binop($1, Exponent, $3) }
  | ID RAISEASSIGN expr { IdAssign($1, Binop(Id($1), Exponent, $3)) }
  | expr RAISE TRANSP { Unop(Transp, $1)  }
  | NOT expr      { Unop(Neg, $2) }
  | PLUSREDUCE expr { Unop(Plusreduce, $2) }
  | MULREDUCE expr { Unop(Mulreduce, $2)  }
  | expr LBRACK expr_list RBRACK { Selection($1, $3)}
  | expr ASSIGN expr { Assign($1, $3)      }
  | ID LPAREN args_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2              }
  | VLINE expr VLINE { Unop(Size, $2)     }
  | LBRACK mat_content RBRACK { UnkMatLit($2) }
  | LBRACK graph_content RBRACK { GraphLit($2) }

mat_content:
    mat_row { [$1] }
  | mat_content SEMI mat_row {$3 :: $1}

mat_row:
    lit { [$1] }
  | mat_row COMMA lit {$3 :: $1 }
  | /* nothing */ {[]}

graph_content:
    edge { [$1] }
  | graph_content SEMI edge {$3 :: $1}

edge:
      INTLITERAL EDGE INTLITERAL { ($1, $3) }

args_opt:
    /* nothing */ { [] }
```

```
  | args_list { List.rev $1 }

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

```ocaml
(* Semantic checking for the BLAStoff compiler *)

open Ast
module StringMap = Map.Make (String)

(* Semantic checking of the AST. Returns an SAST if successful,
   throws an exception if something is wrong.
   Check each global variable, then check each function *)

let check (funcs, stmts) =
  let check_vars loc stmt_lst =
    let add_decl lst = function
      | Expr e ->
        (match e with
        | IdAssign(var, _) -> var :: lst
        | Assign(Id var, _) -> var :: lst
        | _ -> lst)
      | _ -> lst
    in
    let decls = List.fold_left add_decl [] stmt_lst in
    let rec check_dups = function
      | [] -> ()
      | n1 :: n2 :: _ when n1 = n2 -> raise (Failure ("duplicate " ^ n1
          ^ " in " ^ loc))
      | _ :: tl -> check_dups tl
    in
    check_dups (List.sort compare decls)
  in
  (**** Check functions ****)

  (* Collect function declarations for built-in functions: no bodies *)
  let built_in_decls =
    let add_bind map (name, args) =
      StringMap.add name { fname = name; formals = args; body = [] } map
    in
    List.fold_left add_bind StringMap.empty Definitions.functions
  in
  (* Add function name to symbol table *)
  let add_func map fd =
    let built_in_err = "function " ^ fd.fname ^ " may not be defined"
    and dup_err = "duplicate function " ^ fd.fname
    and make_err er = raise (Failure er)
    and n = fd.fname (* Name of the function *) in
```

```
    match fd with
    (* No duplicate functions or redefinitions of built-ins *)
    | _ when StringMap.mem n built_in_decls -> make_err built_in_err
    | _ when StringMap.mem n map -> make_err dup_err
    | _ -> StringMap.add n fd map
  in
  (* Collect all function names into one symbol table *)
  let function_decls = List.fold_left add_func built_in_decls funcs in
  let find_func fname =
    try StringMap.find fname function_decls with
    | Not_found -> raise (Failure ("Undeclared function " ^ fname))
  in
  let is_float = function
    | IntLit _ -> false
    | FloatLit _ -> true
  in
  let contains_float m = List.exists (fun lst -> List.exists is_float
      lst) m in
  let get_char_codes s =
    (* Takes string, returns backwards list of character codes *)
    let rec exp i l = if i < 0 then l else exp (i - 1) (Char.code s.[i]
        :: l) in
    exp (String.length s - 1) []
  in
  let rec check_expr = function
    | Call (fname, args) as call ->
      let fd = find_func fname in
      let num_formals = List.length fd.formals in
      if List.length args != num_formals
      then
        raise
          (Failure
             ("Expecting "
             ^ string_of_int num_formals
             ^ " arguments in "
             ^ string_of_expr call))
      else Call (fname, List.map check_expr args)
    | StringLit s ->
      let chars = List.rev (get_char_codes s) in
      IntMatLit (List.map (fun c -> [ c ]) chars)
    | UnkMatLit m ->
      let has_float = contains_float m in
      (match has_float with
      | true ->
        FloatMatLit
          (List.map
             (fun row ->
               List.map
                 (function
                   | IntLit lit -> float_of_int lit
```

```
                   | FloatLit lit -> lit)
                 row
              ) m)
        | false ->
          IntMatLit
            (List.map
              (fun row ->
                List.map
                  (function
                   | IntLit lit -> lit
                   | FloatLit _ -> raise (Failure "Expected Integers in Matrix"))
                 row)
              m))
      | Id n -> Id n
      | Binop (e1, op, e2) -> Binop (check_expr e1, op, check_expr e2)
      | Unop (op, e) -> Unop (op, check_expr e)
      | FloatMatLit _ -> raise (Failure "Unexpected float matrix in semant checking")
      | IntMatLit _ -> raise (Failure "Unexpected float matrix in semant checking")
      | GraphLit g -> GraphLit g
      | Selection (e, args) -> Selection (check_expr e, List.map check_expr args)
      | IdAssign (n, e) -> IdAssign (n, check_expr e)
      | SelectAssign (n, args, e) -> SelectAssign (n, List.map check_expr args, check_expr e)
      | Assign (e1, e2) ->
        let fix_assign = function
          | Id i, e -> check_expr (IdAssign (i, e))
          | Selection (Id n, args), e -> check_expr (SelectAssign (n, args, e))
          | _ -> raise (Failure "Bad left side of assignment, expected ID or ID[...]")
        in
        fix_assign (e1, e2)
    in
    let rec check_stmt = function
      | Expr e -> Expr (check_expr e)
      | Semiring ring ->
        (match List.mem_assoc ring Definitions.rings with
        | true -> Semiring ring
        | false -> raise (Failure ("Unknown semiring " ^ ring)))
      | Block bl -> Block (check_stmt_list bl)
      | If (p, b1, b2) -> If (check_expr p, check_stmt b1, check_stmt b2)
      | While (p, s) -> While (check_expr p, check_stmt s)
      | Return e -> Return (check_expr e)
    and check_stmt_list = function
      | [ (Return _ as s) ] -> [ check_stmt s ]
      | Return _ :: _ -> raise (Failure "Unreachable statments after
```

```
        return")
    | Block sl :: ss -> check_stmt_list (sl @ ss)
    | s :: ss -> check_stmt s :: check_stmt_list ss
    | [] -> []
  in
  let add_return body =
    match List.rev body with
    | Return _ :: _ -> body
    | _ as l -> List.rev (Return (UnkMatLit [ [] ]) :: l)
  in
  let check_function func =
    let _ = check_vars "body" func.body in
    let checked_body = check_stmt_list (add_return func.body) in
    { fname = func.fname; formals = func.formals; body = checked_body }
  in
  List.map check_function funcs, List.map check_stmt stmts
;;
```

```
module A = Ast
open Ast
open Definitions
module StringMap = Map.Make (String)

let translate (functions, statements) =
  let main_fdecl = { fname = "main"; formals = []; body = List.rev
      statements } in
  let function_decls : (L.llvalue * func_decl) StringMap.t =
    let function_decl m fdecl =
      let name = fdecl.fname
      and formal_types = Array.of_list (List.map (fun _ -> matrix_t)
          fdecl.formals) in
      let ftype = L.function_type matrix_t formal_types in
      StringMap.add name (L.define_function name ftype blastoff_module,
          fdecl) m
    in
    let decls = List.fold_left function_decl StringMap.empty functions in
    StringMap.add
      main_fdecl.fname
      ( L.define_function
          main_fdecl.fname
          (L.function_type i32_t (Array.of_list []))
          blastoff_module
      , main_fdecl )
      decls
  in
  let build_function_body fdecl is_main =
    let func, _ =
      try StringMap.find fdecl.fname function_decls with
      | Not_found -> raise (Failure ("Unknown function, " ^ fdecl.fname))
```

```
  in
  let builder = L.builder_at_end context (L.entry_block func) in
  let local_vars =
    let add_formal m n p =
      L.set_value_name n p;
      let local = L.build_alloca matrix_t n builder in
      ignore (L.build_store p local builder);
      StringMap.add n local m
    in
    let add_local m n =
      if StringMap.mem n m
      then m
      else (
        let local_var = L.build_alloca matrix_t n builder in
        StringMap.add n local_var m)
    in
    let formals =
      List.fold_left2
        add_formal
        StringMap.empty
        fdecl.formals
        (Array.to_list (L.params func))
    in
    let rec add_assignment lst = function
      | Expr e ->
        (match e with
        | IdAssign (id, _) -> id :: lst
        | _ -> lst)
      | Block stmts -> List.fold_left add_assignment lst stmts
      | If (_, s1, s2) -> add_assignment (add_assignment lst s1) s2
      | While (_, s) -> add_assignment lst s
      | _ -> lst
    in
    let locals = List.fold_left add_assignment [] fdecl.body in
    List.fold_left add_local formals locals
  in
  let lookup n =
    try StringMap.find n local_vars with
    | Not_found -> raise (Failure ("Undeclared variable " ^ n))
  in
  let add_terminal builder instr =
    match L.block_terminator (L.insertion_block builder) with
    | Some _ -> ()
    | None -> ignore (instr builder)
  in
  let build_graph_matrix builder m =
    let max3 a b c =
      if a >= b && a >= c then a else if b >= c && b >= a then b else c
    in
    let dim = 1 + List.fold_left (fun acc elem -> max3 acc (fst elem)
```

```
          (snd elem)) 0 m in
      let mat =
        L.build_call
          matrix_create_f
          [| L.const_int i32_t dim; L.const_int i32_t dim |]
          "matrix_create"
          builder
      in
      List.iter
        (fun elem ->
          ignore
            (L.build_call
               matrix_setelem_f
               [| mat
                ; L.const_int i32_t 1
                ; L.const_int i32_t (fst elem)
                ; L.const_int i32_t (snd elem)
               |]
               "matrix_setelem"
               builder))
        m;
      mat
    in
    let build_matrix typ builder m =
      let mat =
        L.build_call
          matrix_create_f
          [| L.const_int i32_t (List.length m)
           ; L.const_int i32_t (List.length (List.hd m))
          |]
          "matrix_create"
          builder
      in
      List.iteri
        (fun i row ->
          (List.iteri (fun j elem ->
               ignore
                 (L.build_call
                    matrix_setelem_f
                    [| mat
                     ; typ elem
                     ; L.const_int i32_t i
                     ; L.const_int i32_t j
                    |]
                    "matrix_setelem"
                    builder)))
            (List.rev row))
        (List.rev m);
      mat
    in
```

```
  let rec fill_select_args builder args =
    let zero =
      L.build_call
        matrix_create_f
        [| L.const_int i32_t 1; L.const_int i32_t 1 |]
        "matrix_create"
        builder
    in
    let base =
      L.build_call
        matrix_create_f
        [| L.const_int i32_t 1; L.const_int i32_t 1 |]
        "matrix_create"
        builder
    in
    let one =
      ignore
        (L.build_call
           matrix_setelem_f
           [| base; L.const_int i32_t 1; L.const_int i32_t 0;
               L.const_int i32_t 0 |]
           "matrix_setelem"
           builder);
      base
    in
    match args with
    | [ _; _; _; _ ] as l -> l
    | [ _; _; _ ] as l -> fill_select_args builder (one :: l)
    | [ _; _ ] as l -> fill_select_args builder (one :: l)
    | [ _ ] as l -> fill_select_args builder (zero :: l)
    | _ -> raise (Failure "Too many/few arguments to selection")
  in
  let rec build_expr builder e =
    match e with
    | IntMatLit m -> build_matrix (fun el -> L.const_int i32_t el)
        builder m
    | GraphLit m -> build_graph_matrix builder m
    | FloatMatLit m -> build_matrix (fun el -> L.const_float float_t
        el) builder m
    | IdAssign (v, e) ->
      let comp_e = build_expr builder e in
      (match v with
      | s -> ignore (L.build_store comp_e (lookup s) builder));
      comp_e
    | Call (fname, exprs) ->
      (match fname with
      | "print" ->
        (match exprs with
        | [ e ] ->
            build_call "matrix_print" [| build_expr builder e |] builder
```

```
        | _ -> raise (Failure "Invalid list of expressions passed to print"))
    | "toString" ->
      (match exprs with
      | [ e ] ->
        build_call
          "matrix_tostring"
          [| build_expr builder e |]
          builder
      | _ -> raise (Failure "Invalid list of expressions passed to toString"))
    | "I" ->
      (match exprs with
      | [ e ] ->
        build_call
          "matrix_create_identity"
          [| build_expr builder e |]
          builder
      | _ -> raise (Failure "Invalid list of expressions passed to I"))
    | "Zero" ->
      (match exprs with
      | [ e ] ->
        build_call "matrix_create_zero" [| build_expr builder e |] builder
      | _ -> raise (Failure "Invalid list of expressions passed to Zero"))
    | "range" ->
      (match exprs with
      | [ e ] ->
        build_call "matrix_create_range" [| build_expr builder e |] builder
      | _ -> raise (Failure "Invalid list of expressions passed to range"))
    | "__ring_push" ->
      (match exprs with
      | [] -> L.build_call ring_push_f [||] "__ring_push" builder
      | _ -> raise (Failure "Invalid list of expressions passed to __ring_push"))
    | "__ring_pop" ->
      (match exprs with
      | [] -> L.build_call ring_pop_f [||] "__ring_pop" builder
      | _ -> raise (Failure "Invalid list of expressions passed to __ring_pop"))
    | f ->
      let fdef, fdecl =
        try StringMap.find f function_decls with
        | Not_found ->
          raise (Failure ("Undeclared function, " ^ f ^ ", found in code generation"))
```

```
        in
        let args = List.map (build_expr builder) (List.rev exprs) in
        L.build_call fdef (Array.of_list args) (fdecl.fname ^
            "_result") builder)
    | Binop (e1, op, e2) ->
      let e1' = build_expr builder e1
      and e2' = build_expr builder e2 in
      (match op with
      | A.Matmul -> build_call "matrix_mul" [| e1'; e2' |] builder
      | A.Exponent -> L.build_call matrix_exp_f [| e1'; e2' |]
          "matrix_mul" builder
      | A.Conv -> build_call "matrix_conv" [| e1'; e2' |] builder
      | A.Elmul -> build_call "matrix_elmul" [| e1'; e2' |] builder
      | A.Add -> build_call "matrix_eladd" [| e1'; e2' |] builder
      | A.Concat -> build_call "matrix_concat" [| e1'; e2' |] builder
      | A.Equal -> build_call "matrix_eq" [| e1'; e2' |] builder
      | A.Neq -> build_call "matrix_neq" [| e1'; e2' |] builder
      | A.Leq -> build_call "matrix_leq" [| e1'; e2' |] builder
      | A.Less -> build_call "matrix_less" [| e1'; e2' |] builder
      | A.Geq -> build_call "matrix_geq" [| e1'; e2' |] builder
      | A.Greater ->
        build_call "matrix_greater" [| e1'; e2' |] builder)
    | UnkMatLit _ -> raise (Failure "Type of matrix is unknown")
    | Assign _ -> raise (Failure "Assign in codegen")
    | StringLit _ -> raise (Failure "StringLit in codegen")
    | Unop (op, e) ->
      let e' = build_expr builder e in
      (match op with
      | A.Size -> build_call "matrix_size" [| e' |] builder
      | A.Transp -> build_call "matrix_transpose" [| e' |] builder
      | A.Plusreduce ->
        build_call
          "matrix_reduce"
          [| e'; L.const_int i32_t 0 |]
          builder
      | A.Mulreduce ->
        build_call
          "matrix_reduce"
          [| e'; L.const_int i32_t 1 |]
          builder
      | A.Neg -> build_call "matrix_negate" [| e' |] builder)
    | Id v -> L.build_load (lookup v) v builder
    | Selection (e, args) ->
      let partialargs' = List.map (build_expr builder) args in
      let filledargs' = fill_select_args builder partialargs' in
      let revfilledargs' = List.rev filledargs' in
      let e' = build_expr builder e in
      let args' = e' :: revfilledargs' in
      L.build_call matrix_extract_f (Array.of_list args')
          "matrix_extract" builder
```

```
    | SelectAssign (v, args, e) ->
      let partialargs' = List.map (build_expr builder) args in
      let filledargs' = fill_select_args builder partialargs' in
      let revfilledargs' = List.rev filledargs' in
      let e' = build_expr builder e in
      let v' = L.build_load (lookup v) v builder in
      let args' = v' :: e' :: revfilledargs' in
      build_call "matrix_insert" (Array.of_list args') builder
  in
  let rec build_stmt builder = function
    | Block sl -> List.fold_left build_stmt builder sl
    | Semiring ring ->
      ignore
        (L.build_call
           ring_change_f
           [| L.const_int i32_t (List.assoc ring Definitions.rings) |]
           "ring_change"
           builder);
      builder
    | Expr e ->
      ignore (build_expr builder e);
      builder
    | Return e ->
      ignore (build_expr builder (Call ("__ring_pop", [])));
      ignore (L.build_ret (build_expr builder e) builder);
      builder
    | If (pred, thn, els) ->
      let pred_expr = build_expr builder pred in
      let mat_truthiness =
        L.build_call matrix_truthy_f [| pred_expr |] "matrix_truthy"
            builder
      in
      let bool_val =
        L.build_icmp L.Icmp.Eq mat_truthiness (L.const_int i32_t 1)
            "i1_t" builder
      in
      let merge_bb = L.append_block context "merge_if" func in
      let build_br_merge = L.build_br merge_bb in
      let then_bb = L.append_block context "then" func in
      add_terminal (build_stmt (L.builder_at_end context then_bb) thn)
          build_br_merge;
      let else_bb = L.append_block context "else" func in
      add_terminal (build_stmt (L.builder_at_end context else_bb) els)
          build_br_merge;
      ignore (L.build_cond_br bool_val then_bb else_bb builder);
      L.builder_at_end context merge_bb
    | While (pred, body) ->
      let pred_bb = L.append_block context "while" func in
      let pred_builder = L.builder_at_end context pred_bb in
      let pred_expr = build_expr pred_builder pred in
```

```
        let mat_truthiness =
          L.build_call matrix_truthy_f [| pred_expr |] "matrix_truthy" pred_builder
        in
        let bool_val =
          L.build_icmp L.Icmp.Eq mat_truthiness (L.const_int i32_t 1) "i1_t" pred_builder
        in
        ignore (L.build_br pred_bb builder) (* builds branch to while from entry point *);
        let body_bb = L.append_block context "while_body" func in
        let body_builder = build_stmt (L.builder_at_end context body_bb) body in
        add_terminal body_builder (L.build_br pred_bb);
        let merge_bb = L.append_block context "merge" func in
        ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
        L.builder_at_end context merge_bb
    in
    let body = Expr (Call ("__ring_push", [])) :: fdecl.body in
    let builder = build_stmt builder (Block body) in
    add_terminal
      builder
      (L.build_ret (L.const_int (if is_main then i32_t else matrix_t) 0))
  in
  build_function_body main_fdecl true;
  List.iter2 build_function_body functions (List.map (fun _ -> false) functions);
  blastoff_module
;;
```

```
module L = Llvm

let context = L.global_context ()
let llmem = L.MemoryBuffer.of_file "graphblas.bc"
let llm = Llvm_bitreader.parse_bitcode context llmem
let blastoff_module = L.create_module context "BLAStoff"
let rings = [ "_", 0; "arithmetic", 1; "logical", 2; "maxmin", 3 ]

let functions =
  [ "I", [ "n" ]
  ; "Zero", [ "d" ]
  ; "range", [ "n" ]
  ; "print", [ "e" ]
  ; "toString", [ "e" ]
  ]
;;

type built_in =
  { name : string
```

```
  ; ret : L.lltype
  ; args : L.lltype list
  }

let i32_t = L.i32_type context
let float_t = L.double_type context

let matrix_t =
  L.pointer_type
    (match L.type_by_name llm "struct.matrix" with
    | None -> raise (Failure "matrix type implementation not found")
    | Some t -> t)
;;

let built_in_defs : built_in list =
  [ { name = "matrix_create"; ret = matrix_t; args = [ matrix_t ] }
  ; { name = "matrix_create_identity"; ret = matrix_t; args = [ matrix_t
      ] }
  ; { name = "matrix_create_zero"; ret = matrix_t; args = [ matrix_t ] }
  ; { name = "matrix_create_range"; ret = matrix_t; args = [ matrix_t ] }
  ; { name = "matrix_print"; ret = matrix_t; args = [ matrix_t ] }
  ; { name = "matrix_tostring"; ret = matrix_t; args = [ matrix_t ] }
  ; { name = "change_ring"; ret = i32_t; args = [ i32_t ] }
  ; { name = "matrix_setelem"; ret = i32_t; args = [ matrix_t; i32_t;
      i32_t; i32_t ] }
  ; { name = "matrix_mul"; ret = matrix_t; args = [ matrix_t; matrix_t ]
      }
  ; { name = "matrix_conv"; ret = matrix_t; args = [ matrix_t; matrix_t
      ] }
  ; { name = "matrix_elmul"; ret = matrix_t; args = [ matrix_t; matrix_t
      ] }
  ; { name = "matrix_eladd"; ret = matrix_t; args = [ matrix_t; matrix_t
      ] }
  ; { name = "matrix_extract"
    ; ret = matrix_t
    ; args = [ matrix_t; matrix_t; matrix_t; matrix_t; matrix_t ]
    }
  ; { name = "matrix_insert"
    ; ret = matrix_t
    ; args = [ matrix_t; matrix_t; matrix_t; matrix_t; matrix_t;
        matrix_t ]
    }
  ;{name = "matrix_eq"; ret = matrix_t; args = [ matrix_t; matrix_t ]}
  ;{name = "matrix_neq"; ret = matrix_t; args = [ matrix_t; matrix_t ]}
  ;{name = "matrix_leq"; ret = matrix_t; args = [ matrix_t; matrix_t ]}
  ;{name = "matrix_less"; ret = matrix_t; args = [ matrix_t; matrix_t ]}
  ;{name = "matrix_geq"; ret = matrix_t; args = [ matrix_t; matrix_t ]}
  ;{name = "matrix_greater"; ret = matrix_t; args = [ matrix_t; matrix_t
      ]}
  ;{name = "matrix_concat"; ret = matrix_t; args = [ matrix_t; matrix_t
```

```
      ]}
    ;{ name = "matrix_bool"; ret = i32_t; args = [ matrix_t ] }
    ;{ name = "matrix_negate"; ret = matrix_t; args = [ matrix_t ] }
    ;{ name = "matrix_reduce"; ret = matrix_t; args = [ matrix_t ; i32_t] }
    ;{ name = "matrix_insert"; ret = matrix_t; args = [ matrix_t;
      matrix_t; matrix_t; matrix_t; matrix_t; matrix_t] }
    ;{ name = "matrix_reduce"; ret = matrix_t; args = [ matrix_t ; i32_t] }
    ;{ name = "matrix_size"; ret = matrix_t; args = [ matrix_t ] }
    ;{ name = "matrix_transpose"; ret = matrix_t; args = [ matrix_t ] }
    ;{ name = "matrix_truthy"; ret = matrix_t; args = [ matrix_t ] }
    ]
;;

let matrix_truthy_t = L.function_type i32_t [| matrix_t |]
let matrix_truthy_f = L.declare_function "matrix_truthy" matrix_truthy_t
    blastoff_module


let matrix_exp_t = L.function_type matrix_t [| matrix_t; matrix_t |]
let matrix_exp_f = L.declare_function "matrix_exp" matrix_exp_t
    blastoff_module

let create_fun_type fdef = L.function_type fdef.ret (Array.of_list
    fdef.args)
let declare_fun fname ftype = L.declare_function fname ftype
    blastoff_module
let built_ins = List.map (fun fdef -> fdef.name, declare_fun fdef.name
    (create_fun_type fdef)) built_in_defs
let build_call fname args builder = L.build_call (List.assoc fname
    built_ins) args fname builder

let matrix_create_t = L.function_type matrix_t [| i32_t; i32_t |]
let matrix_create_f = L.declare_function "matrix_create" matrix_create_t
    blastoff_module
let matrix_identity_t = L.function_type matrix_t [| matrix_t |]
let matrix_identity_f =
  L.declare_function "matrix_create_identity" matrix_identity_t
      blastoff_module
let ring_push_t = L.function_type i32_t [||]
let ring_push_f = L.declare_function "ring_push" ring_push_t
    blastoff_module
let ring_pop_t = L.function_type i32_t [||]
let ring_pop_f = L.declare_function "ring_pop" ring_pop_t blastoff_module
let ring_change_t = L.function_type i32_t [| i32_t |]
let ring_change_f = L.declare_function "ring_change" ring_change_t
    blastoff_module
let matrix_setelem_t = L.function_type i32_t [| matrix_t; i32_t; i32_t;
    i32_t |]
let matrix_setelem_f =
  L.declare_function "matrix_setelem" matrix_setelem_t blastoff_module
```

```
let matrix_extract_t =
  L.function_type matrix_t [| matrix_t; matrix_t; matrix_t; matrix_t;
      matrix_t |]
let matrix_extract_f =
  L.declare_function "matrix_extract" matrix_extract_t blastoff_module
```

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <GraphBLAS.h>

struct matrix {
    GrB_Matrix mat;
};

static void die(const char *msg)
{
    if (errno)
        perror(msg);
    else
        fprintf(stderr, "%s\n", msg);
    exit(1);
}

#define GrB_die(msg, object)        \
do {                                \
    const char *GrB_msg;            \
    GrB_error(&GrB_msg, object);    \
    fprintf(stderr, "%s\n", GrB_msg); \
    die(msg);                       \
} while (0)

static int GrB_ok(GrB_Info info)
{
    if (info == GrB_SUCCESS || info == GrB_NO_VALUE) {
        return 1;
    } else {
        fprintf(stderr, "GrB_ok saw error code: %d\n", info);
        return 0;
    }
}

void GrB_print(GrB_Matrix mat)
{
    if (!GrB_ok(GxB_Matrix_fprint(mat, NULL, GxB_COMPLETE_VERBOSE,
         stdout)))
        die("GxB_Matrix_fprint");
}

```

```c
void GrB_size(GrB_Matrix mat, GrB_Index *nrows, GrB_Index *ncols)
{
    if (nrows && !GrB_ok(GrB_Matrix_nrows(nrows, mat)))
        GrB_die("GrB_Matrix_nrows", mat);

    if (ncols && !GrB_ok(GrB_Matrix_ncols(ncols, mat)))
        GrB_die("GrB_Matrix_ncols", mat);
}

int32_t GrB_scalar(GrB_Matrix mat)
{
    GrB_Index nrows, ncols;
    int32_t elem;

    GrB_size(mat, &nrows, &ncols);
    if (nrows != 1 || ncols != 1)
        die("GrB_scalar mat dims bad");

    if (!GrB_ok(GrB_Matrix_extractElement(&elem, mat, 0, 0)))
        GrB_die("GrB_Matrix_extractElement", mat);

    return elem;
}

/* automatically called before main() */
__attribute__((constructor))
static void matrix_lib_init(void) {
    if (!GrB_ok(GrB_init(GrB_NONBLOCKING)))
        die("GrB_init");
}

/* automatically called after main() */
__attribute__((destructor))
void matrix_lib_finalize(void)
{
    if (!GrB_ok(GrB_finalize()))
        die("GrB_finalize");
}

/* BELOW: Functions used externally */

// begin ring_* functions //

// stack of rings, implemented as intrusive linked list
struct ring {
    GrB_Semiring ring;
    struct ring *prev;
};

struct ring *curr_ring = NULL;
```

```c

void ring_push()
{
    struct ring *r = malloc(sizeof(*r));
    r->ring = GrB_PLUS_TIMES_SEMIRING_INT32;
    r->prev = curr_ring;
    curr_ring = r;
}

void ring_pop()
{
    struct ring *prev;

    if (!curr_ring)
        die("ring_change: curr_ring is NULL");

    prev = curr_ring->prev;
    free(curr_ring);
    curr_ring = prev;
}

void ring_change(int which)
{
    if (!curr_ring)
        die("ring_change: curr_ring is NULL");

    if (which == 0) {
        if (!curr_ring->prev)
            die("ring_change to #_ but curr_ring->prev is NULL");
        curr_ring->ring = curr_ring->prev->ring;
    } else if (which == 1) {
        curr_ring->ring = GrB_PLUS_TIMES_SEMIRING_INT32;
    } else if (which == 2) {
        curr_ring->ring = GrB_LAND_LOR_SEMIRING_BOOL;
    } else if (which == 3) {
        curr_ring->ring = GrB_MAX_MIN_SEMIRING_INT32;
    } else {
        die("ring_change: unknown semiring");
    }
}

// end ring_* functions //

// begin matrix_* functions //

int matrix_getelem(struct matrix *A, int row, int col)
{
    int32_t elem = 0;

    if (!GrB_ok(GrB_Matrix_extractElement(&elem, A->mat, row, col)))
```

```
        GrB_die("GrB_Matrix_extractElement", A->mat);

    return elem;
}

void matrix_setelem(struct matrix *A, int val, int row, int col)
{
    // 0 is the implicit value; storing it explicitly would waste space
    int32_t unused;
    if (val == 0 &&
            GrB_Matrix_extractElement(&unused, A->mat, row, col) ==
                GrB_NO_VALUE)
        return;

    if (!GrB_ok(GrB_Matrix_setElement(A->mat, val, row, col)))
        GrB_die("GrB_Matrix_setElement", A->mat);
}


struct matrix *matrix_create(int nrows, int ncols)
{
    struct matrix *A;
    if (!(A = malloc(sizeof *A)))
        die("malloc failed");

    if (!GrB_ok(GrB_Matrix_new(&A->mat, GrB_INT32, nrows, ncols)))
        GrB_die("GrB_Matrix_new", A->mat);

    return A;
}

struct matrix *matrix_create_zero(struct matrix *dims)
{
    GrB_Index dim_nrows, dim_ncols, nrows, ncols;

    GrB_size(dims->mat, &dim_nrows, &dim_ncols);
    if ((dim_nrows != 1 && dim_nrows != 2) || dim_ncols != 1)
        die("matrix_create_zero invalid dims arg");

    nrows = matrix_getelem(dims, 0, 0);
    ncols = dim_nrows == 2 ? matrix_getelem(dims, 1, 0) : nrows;

    return matrix_create(nrows, ncols);
}

struct matrix *matrix_create_identity(struct matrix *N_scalar)
{
    struct matrix *A;
    GrB_Index i, n;

```

```
    n = GrB_scalar(N_scalar->mat);
    A = matrix_create(n, n);
    for (i = 0; i < n; i++)
        matrix_setelem(A, 1, i, i);

    return A;
}

struct matrix *matrix_create_range(struct matrix *range)
{
    struct matrix *A;
    int32_t lo, hi;
    GrB_Index i, range_nrows, range_ncols;

    GrB_size(range->mat, &range_nrows, &range_ncols);
    if (range_nrows == 1 && range_ncols == 1) {
        lo = 0;
        hi = matrix_getelem(range, 0, 0);
    } else if (range_nrows == 2 && range_ncols == 1) {
        lo = matrix_getelem(range, 0, 0);
        hi = matrix_getelem(range, 1, 0);
    } else {
        die("matrix_create_range invalid range arg");
    }

    if (lo > hi)
        return matrix_create(0, 1);

    A = matrix_create(hi - lo, 1);
    i = 0;
    while (lo < hi)
        matrix_setelem(A, lo++, i++, 0);

    return A;
}

struct matrix *matrix_print(struct matrix *A)
{
    GrB_Index nrows, ncols, i;
    int elem;

    GrB_size(A->mat, &nrows, &ncols);
    if (ncols != 1)
        die("Tried to print string with more than 1 col");

    for (i = 0; i < nrows && (elem = matrix_getelem(A, i, 0)) != 0; i++)
        putchar(elem);

    struct matrix *R = matrix_create(0, 0);
    return R;
```

```
}

struct matrix *matrix_tostring(struct matrix *A)
{
    struct matrix *B;
    GrB_Index nrows, ncols, i, j, k;
    char buf[1000], *b;

    GrB_size(A->mat, &nrows, &ncols);
    B = matrix_create(nrows * (ncols + 1) * 20, 1);

    if (nrows == 0 || ncols == 0)
        return B;

    k = 0;
    for (i = 0; i < nrows; i++) {
        for (j = 0; j < ncols; j++) {
            snprintf(buf, sizeof(buf), "%d ", matrix_getelem(A, i, j));
            for (b = buf; *b; b++)
                matrix_setelem(B, *b, k++, 0);
        }
        matrix_setelem(B, '\n', k++, 0);
    }
    matrix_setelem(B, 0, k, 0);

    return B;
}


struct matrix *matrix_mul(struct matrix *A, struct matrix *B)
{
    struct matrix *C;
    GrB_Info info;
    GrB_Index nrows, ncols, eq1, eq2;

    GrB_size(A->mat, &nrows, &eq1);
    GrB_size(B->mat, &eq2, &ncols);
    if (eq1 != eq2)
        die("matrix_mul bad dimensions");

    C = matrix_create(nrows, ncols);

    info = GrB_mxm(C->mat,
                   GrB_NULL,
                   GrB_NULL,
                   curr_ring->ring,
                   A->mat,
                   B->mat,
                   GrB_NULL);

```

```c
    if (!GrB_ok(info))
        GrB_die("GrB_mxm", A->mat);

    return C;
}

struct matrix *matrix_exp(struct matrix *A, struct matrix *N_scalar)
{
    struct matrix *B;
    int n;
    GrB_Index i, nrows, ncols;

    GrB_size(A->mat, &nrows, &ncols);
    if (nrows != ncols)
        die("matrix_exp mat not square");

    n = GrB_scalar(N_scalar->mat);
    if (n < 1)
        die("matrix_exp needs positive exponent");

    B = A;
    for (i = 0; i < n - 1; i++) {
        B = matrix_mul(A, B);
    }

    return B;
}

struct matrix *matrix_elmul(struct matrix *A, struct matrix *B)
{
    struct matrix *C;
    GrB_Info info;
    GrB_Index A_nrows, A_ncols, B_nrows, B_ncols;

    GrB_size(A->mat, &A_nrows, &A_ncols);
    GrB_size(B->mat, &B_nrows, &B_ncols);

    if (A_nrows != B_nrows || A_ncols != B_ncols)
        die("matrix_elmul bad dimensions");

    C = matrix_create(A_nrows, A_ncols);

    info = GrB_Matrix_eWiseMult_Semiring(C->mat,
                                         GrB_NULL,
                                         GrB_NULL,
                                         curr_ring->ring,
                                         A->mat,
                                         B->mat,
                                         GrB_NULL);

```

```c
    if (!GrB_ok(info))
        GrB_die("GrB_Matrix_eWiseMult_Semiring", A->mat);

    return C;
}

struct matrix *matrix_eladd(struct matrix *A, struct matrix *B)
{
    struct matrix *C;
    GrB_Info info;
    GrB_Index A_nrows, A_ncols, B_nrows, B_ncols;

    GrB_size(A->mat, &A_nrows, &A_ncols);
    GrB_size(B->mat, &B_nrows, &B_ncols);

    if (A_nrows != B_nrows || A_ncols != B_ncols)
        die("matrix_eladd bad dimensions");

    C = matrix_create(A_nrows, A_ncols);

    info = GrB_Matrix_eWiseAdd_Semiring(C->mat,
                                        GrB_NULL,
                                        GrB_NULL,
                                        curr_ring->ring,
                                        A->mat,
                                        B->mat,
                                        GrB_NULL);

    if (!GrB_ok(info))
        GrB_die("GrB_Matrix_eWiseAdd_Semiring", A->mat);

    return C;
}

struct matrix *matrix_extract(struct matrix *M, struct matrix *A, struct
     matrix *B, struct matrix *C, struct matrix *D)
{
    struct matrix *R;
    GrB_Index A_nrows, A_ncols, B_nrows, B_ncols, C_nrows, C_ncols,
         D_nrows, D_ncols;
    int i, j, v, w;

    // verify that A, B, C, D are all integer matrices??

    //veryify that A, B are column vectors and that C, D are 1x1

    GrB_size(A->mat, &A_nrows, &A_ncols);
    GrB_size(B->mat, &B_nrows, &B_ncols);
    GrB_size(C->mat, &C_nrows, &C_ncols);
    GrB_size(D->mat, &D_nrows, &D_ncols);
```

```
    if (A_ncols != 1 || B_ncols != 1 || C_nrows != 1 || C_ncols != 1 || D_nrows != 1 || D_ncols != 1)
        die("matrix_extract bad dimensions");

    int cval = matrix_getelem(C, 0, 0);
    int dval = matrix_getelem(D, 0, 0);
    R = matrix_create(A_nrows*cval, B_nrows*dval);

    //(A[i], B[j]) is top-left corner in form (cols, rows)
    //(A[i]+v, B[j]+w) is what we iterate through
    //(i*cval+v, j*dval+w) is where we store
    for (i = 0; i < A_nrows; i++){
      for (j = 0; j < B_nrows; j++){
        int Ai = matrix_getelem(A, i, 0);
        int Bj = matrix_getelem(B, j, 0);
        for (v = 0; v < cval; v++){
          for (w = 0; w < dval; w++){
              matrix_setelem(R, matrix_getelem(M, Ai+v, Bj+w), i*cval+v, j*dval+w);
          }
        }
      }
    }

    return R;
}

struct matrix *matrix_insert(struct matrix *M, struct matrix *N, struct matrix *A, struct matrix *B, struct matrix *C, struct matrix *D)
{
    //Syntax is like M[A,B,C,D] = N;
    GrB_Index A_nrows, A_ncols, B_nrows, B_ncols, C_nrows, C_ncols, D_nrows, D_ncols, N_nrows, N_ncols;
    int i, j, v, w;

    // verify that A, B, C, D are all integer matrices??

    //veryify that A, B are column vectors and that C, D are 1x1

    GrB_size(A->mat, &A_nrows, &A_ncols);
    GrB_size(B->mat, &B_nrows, &B_ncols);
    GrB_size(C->mat, &C_nrows, &C_ncols);
    GrB_size(D->mat, &D_nrows, &D_ncols);
    GrB_size(N->mat, &N_nrows, &N_ncols);

    if (A_ncols != 1 || B_ncols != 1 || C_ncols != 1 || C_nrows != 1 || D_nrows != 1 || D_ncols != 1)
        die("matrix_extract bad dimensions");

```

```
    int cval = matrix_getelem(C, 0, 0);
    int dval = matrix_getelem(D, 0, 0);

    if ((N_nrows != cval) | (N_ncols != dval))
       die("matrix_extract size mismatch");

    for (i = 0; i < A_nrows; i++){
      for (j = 0; j < B_nrows; j++){
        int Ai = matrix_getelem(A, i, 0);
        int Bj = matrix_getelem(B, j, 0);
        for (v = 0; v < cval; v++){
          for (w = 0; w < dval; w++){
              matrix_setelem(M, matrix_getelem(N, v, w), Ai+v, Bj+w);
          }
        }
      }
    }

    return N;
}

struct matrix *matrix_size(struct matrix *A)
{
    struct matrix *S;
    GrB_Index nrows, ncols;
    GrB_size(A->mat, &nrows, &ncols);

    S = matrix_create(2,1);

    matrix_setelem(S, nrows, 0, 0);
    matrix_setelem(S, ncols, 1, 0);

    return S;
}

struct matrix *matrix_reduce(struct matrix *A, int mult_flag)
{
    struct matrix *R;
    GrB_Index nrows;
    GrB_size(A->mat, &nrows, NULL);

    GrB_Vector v;
    GrB_Vector_new(&v, GrB_INT32, nrows) ;

    GrB_Monoid op;

    if(mult_flag){
      GrB_BinaryOp mult;
      GxB_Semiring_multiply(&mult, curr_ring->ring);
      // TODO: Find a better way of doing mutliplicative identity
```

```c
      GrB_Monoid_new_INT32(&op, mult, 0);
    } else {
      GxB_Semiring_add(&op, curr_ring->ring);
    }

    GrB_Matrix_reduce_Monoid(v, GrB_NULL, GrB_NULL, op, A->mat,
        GrB_NULL);

    R = matrix_create(nrows,1);
    GrB_Col_assign(R->mat, GrB_NULL, GrB_NULL, v, GrB_ALL, nrows, 0,
        GrB_NULL);

    return R;
}

struct matrix *matrix_transpose(struct matrix *A)
{
    struct matrix *T;
    GrB_Index nrows, ncols;
    GrB_size(A->mat, &nrows, &ncols);

    T = matrix_create(ncols, nrows);
    GrB_transpose(T->mat, GrB_NULL, GrB_NULL, A->mat, GrB_NULL);

    return T;
}

struct matrix *matrix_negate(struct matrix *A)
{
    struct matrix *R;
    GrB_Index nrows, ncols;
    int i,j;
    GrB_size(A->mat, &nrows, &ncols);

    R = matrix_create(nrows, ncols);

    for (i = 0; i < nrows; i++) {
        for (j = 0; j < ncols; j++) {
          matrix_setelem(R, matrix_getelem(A, i, j) == 0, i, j);
        }
    }

    return R;
}

struct matrix *matrix_conv(struct matrix *A, struct matrix *B)
{
    struct matrix *C;
    struct matrix *E;
    struct matrix *f;
```

```c
    struct matrix *g;
    struct matrix *h;
    GrB_Index A_nrows, A_ncols, B_nrows, B_ncols, C_nrows, C_ncols;
    int i, j;

    GrB_size(A->mat, &A_nrows, &A_ncols);
    GrB_size(B->mat, &B_nrows, &B_ncols);

    if (A_nrows < B_nrows || A_ncols < B_ncols)
        die("matrix_conv bad dimensions");

    // lots of memory leaked here!

    GrB_Index *row_indices, *col_indices;
    if (!(row_indices = malloc(B_nrows * sizeof(int)))) die("malloc
        failed");
    if (!(col_indices = malloc(B_ncols * sizeof(int)))) die("malloc
        failed");

    C_nrows = A_nrows - B_nrows + 1;
    C_ncols = A_ncols - B_ncols + 1;
    C = matrix_create(C_nrows, C_ncols);
    E = matrix_create(B_nrows, B_ncols);
    f = matrix_create(B_nrows, 1);
    g = matrix_create(1, B_nrows);
    h = matrix_create(1, 1);

    for (i = 0; i < C_nrows; i++) {
        for (j = 0; j < C_ncols; j++) {
          int k;
          for (k = 0; k < B_nrows; k++) row_indices[k] = i+k;
          for (k = 0; k < B_ncols; k++) col_indices[k] = j+k;
          GrB_extract(E->mat, GrB_NULL, GrB_NULL, A->mat, row_indices,
              B_nrows, col_indices, B_ncols, GrB_NULL);
          E = matrix_elmul(E, B);
          f = matrix_reduce(E, 0);
          g = matrix_transpose(f);
          h = matrix_reduce(g, 0);
          matrix_setelem(C, matrix_getelem(h, 0, 0), i, j);
        }
    }

    return C;
}

struct matrix *matrix_concat(struct matrix *A, struct matrix *B)
{
    struct matrix *C;
    GrB_Info info;
    GrB_Index A_nrows, A_ncols, B_nrows, B_ncols;
```

```c
    int i;

    GrB_size(A->mat, &A_nrows, &A_ncols);
    GrB_size(B->mat, &B_nrows, &B_ncols);

    if (A_ncols != B_ncols)
        die("matrix_concat bad dimensions");

    GrB_Index *A_row_indices, *B_row_indices, *col_indices;
    if (!(A_row_indices = malloc(A_nrows * sizeof(int)))) die("malloc
        failed");
    if (!(B_row_indices = malloc(B_nrows * sizeof(int)))) die("malloc
        failed");
    if (!(col_indices = malloc(A_ncols * sizeof(int)))) die("malloc
        failed");

    for (i = 0; i < A_nrows; i++) A_row_indices[i] = i;
    for (i = A_nrows; i < A_nrows + B_nrows; i++) B_row_indices[i -
        A_nrows] = i;
    for (i = 0; i < A_ncols; i++) col_indices[i] = i;

    C = matrix_create(A_nrows + B_nrows, A_ncols);

    info = GrB_assign(C->mat,
                      GrB_NULL,
                      GrB_NULL,
                      A->mat,
                      A_row_indices,
                      A_nrows,
                      GrB_ALL,
                      A_ncols,
                      GrB_NULL);

    info = GrB_assign(C->mat,
                      GrB_NULL,
                      GrB_NULL,
                      B->mat,
                      B_row_indices,
                      B_nrows,
                      GrB_ALL,
                      B_ncols,
                      GrB_NULL);

    if (!GrB_ok(info))
        GrB_die("GrB_Matrix_eWiseAdd_Semiring", A->mat);

    return C;
}

// Comparison operators
```

```
struct matrix *matrix_elcompare(struct matrix *A, struct matrix *B, int
    op_index)
{
    struct matrix *C;
    int i, j;
    int a, b, comp_val;

    GrB_Index nrows, ncols, nrowsB, ncolsB;
    GrB_size(A->mat, &nrows, &ncols);
    GrB_size(B->mat, &nrowsB, &ncolsB);

    /*
    printf("dims of A: %d %d\n", (int) nrows, (int) ncols);
    matrix_print(matrix_tostring(A));
    printf("dims of B: %d %d\n", (int) nrowsB, (int) ncolsB);
    matrix_print(matrix_tostring(B));
    */

    C = matrix_create(1, 1);

    if (nrows != nrowsB || ncols != ncolsB)
        die("Can't compare two matrices that are different dimensions");

    for (i = 0; i < nrows; i++) {
        for (j = 0; j < ncols; j++) {
            a = matrix_getelem(A, i, j);
            b = matrix_getelem(B, i, j);
            switch (op_index) {
                case 0: comp_val = a == b; break;
                case 1: comp_val = a != b; break;
                case 2: comp_val = a <= b; break;
                case 3: comp_val = a < b; break;
                case 4: comp_val = a >= b; break;
                case 5: comp_val = a > b; break;
                default: die("Unknown comparison operator");
            }
            if (!comp_val) return C;
        }
    }
    matrix_setelem(C, 1, 0, 0);
    return C;
}

struct matrix *matrix_eq(struct matrix *A, struct matrix *B) { return
    matrix_elcompare(A, B, 0); }
struct matrix *matrix_neq(struct matrix *A, struct matrix *B) { return
    matrix_elcompare(A, B, 1); }
struct matrix *matrix_leq(struct matrix *A, struct matrix *B) { return
    matrix_elcompare(A, B, 2); }
```

```
struct matrix *matrix_less(struct matrix *A, struct matrix *B) { return matrix_elcompare(A, B, 3); }
struct matrix *matrix_geq(struct matrix *A, struct matrix *B) { return matrix_elcompare(A, B, 4); }
struct matrix *matrix_greater(struct matrix *A, struct matrix *B) { return matrix_elcompare(A, B, 5); }

// "The truth value of an expr is equivalent to expr > 0" (Jake, 2021)
int matrix_truthy(struct matrix *A)
{
    struct matrix *C;
    struct matrix *B;
    GrB_Index nrows, ncols;
    GrB_size(A->mat, &nrows, &ncols);

    B = matrix_create(nrows, ncols);
    C = matrix_greater(A, B);

    return matrix_getelem(C, 0, 0) > 0;
}

// end matrix_* functions //

#ifdef RUN_TEST
int main(int argc, char **argv){
    struct matrix *A, *B, *C;

    ring_push();

    A = matrix_create(2, 2);
    B = matrix_create(2, 2);
    // B = matrix_create(1, 1);
    matrix_setelem(A, 2, 0, 0);
    matrix_setelem(A, 2, 0, 1);
    matrix_setelem(A, 2, 1, 0);
    matrix_setelem(A, 2, 1, 1);
    matrix_setelem(B, 2, 0, 0);
    matrix_setelem(B, 2, 0, 1);
    matrix_setelem(B, 2, 1, 0);
    matrix_setelem(B, 2, 1, 1);
    matrix_print(matrix_tostring(A));
    matrix_print(matrix_tostring(B));

    C = matrix_mul(A, B);
    matrix_print(matrix_tostring(C));
}
#endif
```

References

[KG11] Jeremy Kepner and John Gilbert. *Graph Theory in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, 2011. ISBN: 978-0-89871-990-1. URL: `https://www.google.com/books/edition/Graph_Algorithms_in_the_Language_of_Line/BnezR_6PnxMC`.

[Gil] John Gilbert. *GraphBLAS: Graph Algorithms in the Language of Linear Algebra*. URL: `https://sites.cs.ucsb.edu/~gilbert/talks/Gilbert-27Jun2019.pdf`.