BLAStoff Language Final 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 Lexical Conventions

2.1.1 Assignment

Every variable in BLAStoff is a matrix. A matrix variable is defined in the following way:

```
id = expr;
```

where the left-hand side is an identifier, which can be made up of alphanumeric characters and underscores, beginning with an alphabetic character, and the right-hand side is an expression.

Matrices can be defined five ways: as a matrix literal, as a graph, as a number, with a generator function, or as a string. Below we describe are the 5 corresponding expressions.

2.1.1.1 Matrix Literal Definition

A matrix literal looks as follows:

```
[row;
 row;
 ...]
```

where each `row` looks as follows:

```
num, num, ...
```

where each `num` is a positive or negative integer. Here's an example:

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

```
    0,0,-1];
```

which sets M as the matrix

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

.

In the matrix literal definition, the number of items ins must be the same in every row.

2.1.1.2 Graph Definition

The graph definition looks as follows:

```
[
    (edge | int);
    (edge | int);
    ...
]
```

Each `int` is a non-negative integer (`[0-9]+`), and each edge looks as follows:

```
int -> int
```

Here's an example:

```
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).

2.1.1.3 Number Definition

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

```
num
```

using the Here's an example:

```
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

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

We will discuss in the section on operations how these 1x1 matrices are used to replicate things like scalar multiplication.

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. This is what they look like:

```
Zero(expr)
I(expr)
range(expr | expr, expr)
```

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 matrix of two possible sizes. n can be a 2×1 positive 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 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 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:

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

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

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 String Definition

The final definiton method is as a string. It looks like the following:

```
‘str‘
```

where the `str` is any string sequence. This returns a column vector with the ASCII values of the given string. For instance;

```
A = ‘BLAS‘
```

This code would result in the following matrix:

$$A = \begin{bmatrix} 66 \\ 76 \\ 65 \\ 83 \end{bmatrix}$$

It will be apparent later how this is useful.

2.1.2 Comments

There are two types of comments in BLASToff. Single-line comments are denoted by `//`. Multi-line comments begin with `/*` and end with `*/`. 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 as follows:

```
def id(id, id, ...) {
    stmnt;
    stmnt;
    ...
}
```

In functions, returning is optional. Here is a simple example.

```
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 "None" in BLAStoff, a 0×0 matrix.

2.1.4 If statements

If/else statements, look as follows:

```
if (expr) stmnt ?[else stmnt]
```

For example:

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

The truth value of an `expr` is equivalent to `expr > 0`. The `>` operator will be discussed in full later.

2.1.5 For/While Loops

For and while loops look as follows:

```
for (?expr ; expr ; ?expr) stmnt
while (expr) stmnt
```

For example:

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

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

Though we allow for loops, but they are not usually the ideal paradigm. The selection operator, defined later, 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.

Semirings.png

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

Do we primitives.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	$\mathbf{C}(\mathbf{i}, \mathbf{j}) \odot= \mathbf{A}$
	$\mathbf{w}(\mathbf{i}) \odot= \mathbf{u}$

This is how we implement these operators and some more:

2.1.6.1 Selection

Here is the grammar for the selection operator:

```
expr[expr, expr, expr, expr];
expr[expr, expr]
expr[expr];
```

The BLAStoff selection operator can be applied to any matrix and 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 matrices. c, d are optional and have a default value of $\big[1\big]$. B is also optional and its default value is $\big[0\big]$. 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 = Zero([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. 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 Matrix Multiplication

We now define a number of binary operators. The grammars for these operators all look like

```
expr % expr
```

where % is the given operator.

The matrix multiplication operator * looks like the following:

```
A*B
```

where A is an $l \times m$ matrix and B is an $m \times n$ matrix. The product is an $l \times n$ matrix. This operation works like standard matrix multiplication, so I don't have to spend 2 pages explaining how it works, like I did for selection. Here's an example:

```
A = [1,2;
    1,2;
    1,2;
    1,2]
B = [1,2,3;
    1,2,3;]
C = A*B;
```

This would result in:

$$C = \begin{bmatrix} 3 & 6 & 9 \\ 3 & 6 & 9 \\ 3 & 6 & 9 \\ 3 & 6 & 9 \end{bmatrix}$$

2.1.6.3 Convolution

The convolution operator ~ looks like the following:

```
A~B
```

where A is an $m \times n$ matrix and B is an $o \times p$ matrix such that $m \geq o$, $n \geq p$, and $o, p > 0$. The output is an $(m - o + 1) \times (n - p + 1)$ matrix. It works like normal matrix convolution, where B is the kernel and the output of $A.B$ is the result of sliding the kernel, B, along each row of the matrix A and taking the sum of the element-wise product of the kernel and the sub-matrix it covers. Here is an example:

```
A = [1,2,3;
    4,5,6;
    7,8,9];
B = I(2);
C = A~B;
```

This would result in:

$$C = \begin{bmatrix} 6 & 8 \\ 12 & 14 \end{bmatrix}$$

The convolution operator can be used to achieve some other typical operators in Linear Algebra. For instance, scalar multiplication:

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

This would result in:

$$B = \begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \\ 14 & 16 & 18 \end{bmatrix}$$

Or the dot product:

```
v1 = [1;2];
v2 = [2;3];
u = v1~v2;
```

This would result in:

$$u = \begin{bmatrix} 8 \end{bmatrix}$$

2.1.6.4 Element-wise Multiplication

The element-wise multiplication operator `@` looks like the following:

```
A@B
```

where A and B are both $m \times n$ matrices. The output is also a $m \times n$ matrix. This is standard element-wise multiplication, and is rather straightforward. Example:

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

This would result in:

$$C = \begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix}$$

2.1.6.5 Element-wise Addition

The element-wise addition operator `@` looks like the following:

```
A+B
```

where A and B are both $m \times n$ matrices. The output is also a $m \times n$ matrix. This is standard element-wise addition/matrix addition, and is also rather straightforward. Example:

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

This would result in:

$$C = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

2.1.6.6 Exponentiation

The exponentiation operator `^` looks like one of the following forms:

```
expr^(expr | T)
```

We can say these correspond to

```
A^b
A^T
```

First we will look at the A^b case. In this case, A is an $n \times n$ (square) matrix and b is a 1×1 positive matrix. The output will be an $n \times n$ matrix as well. This operator is normal matrix exponentiation. For example:

```
A = [1,2;
    3,4];
B = A^2;
```

This would result in:

$$B = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

In the A^T case, A is any $m \times n$ matrix, and T is a reserved keyword. This returns the transpose of A, an $n \times m$ matrix. Example:

```
A = [1,2,3;
    4,5,6];
B = A^T;
```

This would result in:

$$B = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

2.1.6.7 Size

The size operator `||` looks like the following:

```
|expr|
```

where the value of the expression, A, is any $m \times n$ matrix and returns the 2×1 matrix/column vector

$$\begin{bmatrix} m \\ n \end{bmatrix}$$

Example:

```
A = [1,2,3;
   4,5,6];
B = |A|;
```

This would result in:

$$B = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

Note that this format is the same as the argument to `Zero`! So, consider the following code:

```
C = Zero(|A|);
```

This would result in C being a matrix of the same size as A, but all zeroes! How convenient!

Of course, if you want to extract the number of rows and columns individually, you can use our selection operator:

```
m = |A|[0];
n = |A|[1];
```

Combining this with another selection operator and the `range` function, we can do things like replace every element in A with an arbitrary number, not just 0:

```
A[range(m), range(n)] = 5;
```

2.1.6.8 Vertical Concatenation

The vertical concatenation operator `:` is another binary operator, and looks like one the following:

```
A:B
```

where A is an $m \times n$ matrix and B is an $l \times n$ matrix. The output will be an $(m + l) \times n$ matrix, that consists of A on top of B. Example:

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

```
C = A:B;
```

This would result in:

$$C = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

2.1.6.9 A note on horizontal concatenation

We do not have horizontal concatenation operator. Why is this? Do we hate the horizontal direction? No, it is because you can easily write an efficient function for horizontal concatenation using vertical concatenation, and we will show that function below. In general, any potential operator that can be written as a function, but doesn't employ for loops heavily, that is just as effective as implementing a primitive, we do not use an operator for, and instead put it in our standard library, discussed below.

(It is also worth noting that you can construct an efficient function for vertical concatenation using horizontal concatenation, but we have to choose one of them, and vertical is preferable as BLAStoff uses column vectors more often than row vectors).

2.1.6.10 Reduce Rows

The reduce rows operator `%`, looks like the following:

```
(+|*)%expr
```

So, the two possible forms are

```
+%A
*%A
```

Here, if A is an $m \times n$ matrix, this will output an $m \times 1$ matrix, a column vector.

If

$$A = \begin{bmatrix} A_{0,0} & A_{0,1} & \dots & A_{0,n-1} \\ A_{1,0} & A_{1,1} & \dots & A_{1,n-1} \\ \vdots & \vdots & \vdots & \vdots \\ A_{m-1,0} & A_{m-1,1} & \dots & A_{m-1,n-1} \end{bmatrix}$$

then

$$+\$A = \begin{bmatrix} \sum_{i=0}^{n-1} A_{0,i} \\ \sum_{i=0}^{n-1} A_{1,i} \\ \vdots \\ \sum_{i=0}^{n-1} A_{m-1,i} \end{bmatrix}$$

and

$$*\$A = \begin{bmatrix} \prod_{i=0}^{n-1} A_{0,i} \\ \prod_{i=0}^{n-1} A_{1,i} \\ \vdots \\ \prod_{i=0}^{n-1} A_{m-1,i} \end{bmatrix}$$

Here's a code example:

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

This would result in:

$$B = \begin{bmatrix} 3 \\ 7 \\ 11 \end{bmatrix}$$

$$C = \begin{bmatrix} 2 \\ 12 \\ 30 \end{bmatrix}$$

2.1.6.11 A note on reduce columns

See 2.6.8.1.

2.1.6.12 Assignment operators

The operator `*=`, used as follows:

```
A*=B;
```

is equivalent to

```
A = A*B;
```

The same is true for the other assignment operators:

```
A~=B;
A@=B;
A+=B;
A^=b;
A:=B;
```

2.1.6.13 Comparisons

The comparison operators, all typical binary operators, can be used as follows:

```
A == B
A != B
A > B
A >= B
A < b
A <= B
```

where A and B are both $m \times n$ matrices. These operations return our version of "true," $\left[1\right]$ if these comparisons are hold element-wise in A and B. That, is $\forall(j,i) \in ([0,m) \times [0,n))$, $A_{j,i} \geq B_{j,i}$, using the `>=` operator as an example. Note that $>$ and $<$ are not anti-symmetric under this definition. The one exception to the element-wise rule is `!=`, which is just logical not on `==`.

2.1.6.14 Semiring redefinition

You may have noticed that though we have defined a number of operations on matrices, when we are actually computing these matrix operations, in our examples the only operators we have actually used on the elements of these matrices are have been standard arithmetic $+$ and $\times$. However, 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;
a + b; //returns 2.1; plus is now minimum
a * b; //returns 3; times is now maximum
a * c; //returns 2.1
```

`#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
```

2.1.6.15 Logical Negation

The final operator is logical negation `!`. It looks as follows:

```
!expr
```

where the value of the `expr`, A, is any $m \times n$ matrix. It outputs an $m \times n$ matrix where each element is logically negated. That is, all zeroes become ones and all non-zeroes become zeroes. Here is an example:

```
A = [1,0;
    0,3];
B = !A;
```

This would result in:

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

This operator's behavior is invariant of the semiring, as do selection, transpose, inverse, vertical concatenation, and size.

2.1.7 Precedence

Below is the precedence table for operators, from highest to lowest:

Operator	Symbol	Associativity
Exponentiation	`^`	Right
Selection	`[]`	Left
Logical Negation	`!`	Right
Reduce Rows	`+%`, `*%`	Right
Vertical Concatenation	`:`	Left
Multiplications/Convolution	`*`, `~`, `@`	Left
Addition	`+`	Left
Comparisons	`<`, `>`, `==`, `<=`, `>=`	Left

2.1.8 Keywords

BLAStoff reserves the following keywords:

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

2.2 More Language Details

2.2.1 Memory

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

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

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

Because we use assign-by-value, each matrix has a reference count of 1, and garbage collection is quite simple; you simply de-allocate all variables declared in a function after the function ends.

2.2.2 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); // error
}
```

2.2.3 Printing

We provide the primitive function `print` that takes in one non-negative column vector, with all values less than 127, and prints the corresponding ASCII characters. As you may suspect, this is a good use of the string matrix definition:

```
print("Hello World!\n");

OUTPUT:
Hello World!
```

We also provide a standard library function `toString` that takes in any matrix and returns a column vector corresponding to the pretty-printed string:

```
A = [1, 2;
    3, 4];
print(toString(A));

OUTPUT:
1 2
3 4
```

2.3 Sample Code

2.3.1 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:

2.3.1.1 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;
}
```

2.3.1.2 Horizontal Concatenation

As we said, we don't include this as an operator because it is quite easy to write as a function using vertical concatentation and transpose:

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

2.3.1.3 Plus/Times Column Reduce

Column reduction follows similarly:

```
def plusColumnReduce(A){
    #_;
    return ((+%A)^T)^T;
```

```
}

def timesColumnReduce(A){
    #_;
    return ((*%A)^T)^T;
}
```

2.3.1.4 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);
}
```

2.3.1.5 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++;
        }
```

```
    }
}
```

2.3.2 Graph Algorithms

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

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

BFS pseudocode.png

Our 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)) {
        #arithmetic;
        depth += 1;
        #logical;
        levels[rangeFromVector(frontier)] = depth;
        mask = !(frontier^T)[Zero(N), 0, 1, N];
        maskedGT @= mask;
        frontier = maskedGT*frontier;
    }
    #arithmetic;
    return levels + (One(|levels|~(-1));
}
```

Let's look at how this code works. 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 mutilates $maskedGT \cdot frontier$. 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, 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!

Table 1: Team Roles

Role	Member
Manager	Katon
Language Guru	Jake
System Architect	Michael
Tester	Jason

Table 2: Timeline

Date	Milestone
Jan 25	Decided on graph/matrix language
Jan 27	Came Across the Work of the GraphBLAS Forum
Feb 17	Established Repo
Feb 23	Completed Initial Scanner and Parser
March 24	Created First Program - Declaring a Matrix and Printing It
March 31	Completed Code for First Operation - Matrix Multiplication
April 3	Added Rigorous Semantic Checking
April 9	Added Code to Distinguish Between Int Matrices and Float Matrices
April 17	Added Graph Literals
April 19	Completed the Majority of the Basic Operators
April 20	Completed Generator Functions
April 20	Completed If and While Loops Using "Truthy" Checking For Non-Zero Elements
April 20	Completed Selection Operator
April 21	Completed Semirings
April 22	Completed Testing For Semiring/Operation Combos
April 23	Completed BFS

3 Project Plan

Workflow:

We used gitbub for issue tracking. Issues were opened during our meetings or by anyone who encountered a new obstacle. As our workflow evolved, we realized that issues should only be closed when tests created to represent the issue were passing. Much of our development was test-driven, creating tests regarding features and then using those tests as both the specification and the metric of progress.

We had weekly synchronous meetings on Saturdays beginning in mid-January continuing through the end of the semester. The meetings were all under an hour, primarily aiming to create consensus regarding design decisions. All other communications were asynchronous, primarily over instant message and comments in github issues. We used Ocamlformatter with the Jane Street profile to unify the code standard.

4 Architectural Design

5 Test Plan

6 Lessons Learned

6.1 Katon

All issues should have a testing component attached to them. The issue isn't resolved until a new test or set of tests that target that specific issue are created and pass. I knew before that code that hasn't been run yet is incorrect. But we found out that code in a language that has never been compiled is pseudo-code. It is important to focus on fundamental issues rather than improving upon the few parts that work. Solving the fundamental issues is the most time consuming but also yields the highest reward. Knowing how every part of the code base works, including those written by someone else, set-in-stone, and ostensibly error-free, is vital for debugging an error in any part of the code base. Pretty printers aren't just a nice demonstration, they are important for debugging and should themselves be tested.

6.2 Michael

6.3 Jake

6.4 Jason

7 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
```

```
36    | Assign of expr * expr
37    | IdAssign of string * expr
38    | SelectAssign of string * expr list * expr
39    | Selection of expr * expr list
40    | Call of string * expr list
41    | StringLit of string
42
43  type stmt =
44    | Semiring of string
45    | Block of stmt list
46    | Expr of expr
47    | Return of expr
48    | If of expr * stmt * stmt
49    | While of expr * stmt
50
51  type func_decl =
52    { fname : string
53    ; formals : string list
54    ; body : stmt list
55    }
56
57  type program = func_decl list * stmt list
58
59  (* Pretty-printing functions *)
60
61  let string_of_op = function
62    | Add -> "+"
63    | Matmul -> "*"
64    | Elmul -> "@"
65    | Conv -> "~"
66    | Equal -> "=="
67    | Neq -> "!="
68    | Less -> "<"
69    | Leq -> "<="
70    | Greater -> ">"
71    | Geq -> ">="
72    | Exponent -> "^"
73    | Concat -> ":"
74  ;;
75
76  let string_of_mat lit_to_string m =
77    let string_of_row row =
78      String.concat "," (List.fold_left (fun acc lit -> lit_to_string lit
          :: acc) [] row)
79    in
80    "["
81    ^ String.concat ";" (List.fold_left (fun acc row -> string_of_row row
        :: acc) [] m)
82    ^ "]"
83  ;;
```

```
84
85  let string_of_graph g =
86    let string_of_edge (v1, v2) = string_of_int v1 ^ "->" ^ string_of_int
          v2 in
87    "[" ^ String.concat ";" (List.map string_of_edge g) ^ "]"
88  ;;
89
90  let rec string_of_expr = function
91    | Id s -> s
92    | Binop (e1, o, e2) ->
93      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
94    | Unop (o, e) -> string_of_e_with_uop e o
95    | Assign (e1, e2) -> string_of_expr e1 ^ " = " ^ string_of_expr e2
96    | IdAssign (s, e) -> s ^ " = " ^ string_of_expr e
97    | Call (f, el) -> f ^ "(" ^ String.concat ", " (List.map
          string_of_expr el) ^ ")"
98    | UnkMatLit m ->
99      string_of_mat
100       (fun lit ->
101         match lit with
102         | IntLit ilit -> string_of_int ilit
103         | FloatLit flit -> string_of_float flit)
104       m
105   | IntMatLit m -> string_of_mat string_of_int m
106   | GraphLit g -> string_of_graph g
107   | StringLit s -> "'" ^ s ^ "'"
108   | FloatMatLit m -> string_of_mat string_of_float m
109   | Selection (e, args) ->
110     string_of_expr e ^ "[" ^ String.concat ", " (List.map string_of_expr
          args) ^ "]"
111   | SelectAssign (s, args, e) ->
112     s
113     ^ "["
114     ^ String.concat ", " (List.map string_of_expr args)
115     ^ "]"
116     ^ " = "
117     ^ string_of_expr e
118
119 and string_of_e_with_uop e =
120   let str_expr = string_of_expr e in
121   function
122   | Neg -> "!" ^ str_expr
123   | Size -> "|" ^ str_expr ^ "|"
124   | Transp -> str_expr ^ "^T"
125   | Plusreduce -> "+%" ^ str_expr
126   | Mulreduce -> "*%" ^ str_expr
127 ;;
128
129 let rec string_of_stmt = function
130   | 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
```

```
17
18 %start program
19 %type <Ast.program> program
20
21 %nonassoc NOELSE
22 %nonassoc ELSE
23 %right ASSIGN PLUSASSIGN ELMULASSIGN CONVASSIGN MATMULASSIGN
       CONCATASSIGN RAISEASSIGN
24 %left EQ NEQ
25 %left LT GT LEQ GEQ
26 %right LBRACK RBRACK
27 %left PLUS
28 %left MATMUL ELMUL
29 %left CONCAT CONV
30 %left RAISE
31 %left EDGE
32 %right PLUSREDUCE MULREDUCE
33 %left TRANSP
34 %right NOT
35 %%
36
37 program:
38   units EOF { (List.rev (fst $1), snd $1) }
39
40 units:
41     /* empty */ { ([], []) }
42     | units fdecl { ($2 :: fst $1 , snd $1) }
43     | units stmt { (fst $1, $2 :: snd $1) }
44
45 fdecl:
46    FDECL ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
47    { { fname = $2;
48        formals = $4;
49        body = List.rev $7 } }
50
51 formals_opt:
52     /* nothing */ { [] }
53   | formal_list  { $1 }
54
55 formal_list:
56     ID                  { [$1]    }
57   | formal_list COMMA ID { $3 :: $1 }
58
59 expr_list:
60     expr                  { [$1]    }
61   | expr_list COMMA expr { $3 :: $1 }
62
63 stmt_list:
64     /* nothing */ { [] }
65   | stmt_list stmt { $2 :: $1 }
```

```
66
67  stmt:
68      expr SEMI                                { Expr $1
                                                }
69    | SEMIRING ID SEMI                         { Semiring $2
                                              }
70    | RETURN ret_opt SEMI                      { Return $2
                                                }
71    | LBRACE stmt_list RBRACE                  { Block(List.rev $2)
                                            }
72    | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([]))
                                           }
73    | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7)
                                             }
74    | WHILE LPAREN expr RPAREN stmt           { While($3, $5)
                                           }
75    | FOR LPAREN stmt expr SEMI expr RPAREN stmt { Block([$3 ; While($4,
          Block([$8 ; Expr($6)]))])}
76
77  ret_opt:
78       /* nothing */ { UnkMatLit([[]]) }
79    | expr         { $1 }
80
81
82  lit:
83      INTLITERAL { IntLit($1) }
84    | FLOATLITERAL { FloatLit($1) }
85
86  expr:
87     lit         { UnkMatLit([[$1]]) }
88    | STRINGLITERAL { StringLit($1) }
89    | ID             { Id($1)               }
90    | expr PLUS  expr { Binop($1, Add, $3)  }
91    | ID PLUSASSIGN expr { IdAssign($1, Binop(Id($1), Add, $3)) }
92    | expr MATMUL expr { Binop($1, Matmul, $3) }
93    | ID MATMULASSIGN expr { IdAssign($1, Binop(Id($1), Matmul, $3)) }
94    | expr ELMUL expr { Binop($1, Elmul, $3) }
95    | ID ELMULASSIGN expr { IdAssign($1, Binop(Id($1), Elmul, $3)) }
96    | expr EQ    expr { Binop($1, Equal, $3) }
97    | expr NEQ   expr { Binop($1, Neq, $3)  }
98    | expr LT    expr { Binop($1, Less, $3) }
99    | expr LEQ   expr { Binop($1, Leq, $3)  }
100   | expr GT    expr { Binop($1, Greater, $3) }
101   | expr GEQ   expr { Binop($1, Geq, $3)  }
102   | expr CONV  expr { Binop($1, Conv, $3) }
103   | ID CONVASSIGN expr { IdAssign($1, Binop(Id($1), Conv, $3)) }
104   | expr CONCAT  expr { Binop($1, Concat, $3)}
105   | ID CONCATASSIGN expr { IdAssign($1, Binop(Id($1), Concat, $3)) }
106   | expr RAISE expr { Binop($1, Exponent, $3) }
107   | 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 }
```

```
(* 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 ->
```

```
14          (match e with
15          | Id var -> var :: lst
16          | _ -> lst)
17        | _ -> lst
18      in
19      let decls = List.fold_left add_decl [] stmt_lst in
20      let rec check_dups = function
21        | [] -> ()
22        | n1 :: n2 :: _ when n1 = n2 -> raise (Failure ("duplicate " ^ n1
              ^ " in " ^ loc))
23        | _ :: tl -> check_dups tl
24      in
25      check_dups (List.sort compare decls)
26    in
27    (**** Check functions ****)
28
29    (* Collect function declarations for built-in functions: no bodies *)
30    let built_in_decls =
31      let add_bind map (name, args) =
32        StringMap.add name { fname = name; formals = args; body = [] } map
33      in
34      List.fold_left add_bind StringMap.empty Definitions.functions
35    in
36    (* Add function name to symbol table *)
37    let add_func map fd =
38      let built_in_err = "function " ^ fd.fname ^ " may not be defined"
39      and dup_err = "duplicate function " ^ fd.fname
40      and make_err er = raise (Failure er)
41      and n = fd.fname (* Name of the function *) in
42      match fd with
43      (* No duplicate functions or redefinitions of built-ins *)
44      | _ when StringMap.mem n built_in_decls -> make_err built_in_err
45      | _ when StringMap.mem n map -> make_err dup_err
46      | _ -> StringMap.add n fd map
47    in
48    (* Collect all function names into one symbol table *)
49    let function_decls = List.fold_left add_func built_in_decls funcs in
50    let find_func fname =
51      try StringMap.find fname function_decls with
52      | Not_found -> raise (Failure ("Undeclared function " ^ fname))
53    in
54    let is_float = function
55      | IntLit _ -> false
56      | FloatLit _ -> true
57    in
58    let contains_float m = List.exists (fun lst -> List.exists is_float
          lst) m in
59    let get_char_codes s =
60      (* Takes string, returns backwards list of character codes *)
61      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
```

```
50          (Array.to_list (L.params func))
51        in
52        let rec add_assignment lst = function
53          | Expr e ->
54            (match e with
55            | IdAssign (id, _) -> id :: lst
56            | _ -> lst)
57          | Block stmts -> List.fold_left add_assignment lst stmts
58          | If (_, s1, s2) -> add_assignment (add_assignment lst s1) s2
59          | While (_, s) -> add_assignment lst s
60          | _ -> lst
61        in
62        let locals = List.fold_left add_assignment [] fdecl.body in
63        List.fold_left add_local formals locals
64      in
65      let lookup n =
66        try StringMap.find n local_vars with
67        | Not_found -> raise (Failure ("Undeclared variable " ^ n))
68      in
69      let add_terminal builder instr =
70        match L.block_terminator (L.insertion_block builder) with
71        | Some _ -> ()
72        | None -> ignore (instr builder)
73      in
74      let build_graph_matrix builder m =
75        let max3 a b c =
76          if a >= b && a >= c then a else if b >= c && b >= a then b else c
77        in
78        let dim = 1 + List.fold_left (fun acc elem -> max3 acc (fst elem)
              (snd elem)) 0 m in
79        let mat =
80          L.build_call
81            matrix_create_f
82            [| L.const_int i32_t dim; L.const_int i32_t dim |]
83            "matrix_create"
84            builder
85        in
86        List.iter
87          (fun elem ->
88            ignore
89              (L.build_call
90                 matrix_setelem_f
91                 [| mat
92                  ; L.const_int i32_t 1
93                  ; L.const_int i32_t (fst elem)
94                  ; L.const_int i32_t (snd elem)
95                 |]
96                 "matrix_setelem"
97                 builder))
98          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 |]
```

```
148             "matrix_setelem"
149             builder);
150         base
151       in
152       match args with
153       | [ _; _; _; _ ] as l -> l
154       | [ _; _; _ ] as l -> fill_select_args builder (one :: l)
155       | [ _; _ ] as l -> fill_select_args builder (one :: l)
156       | [ _ ] as l -> fill_select_args builder (zero :: l)
157       | _ -> raise (Failure "Too many/few arguments to selection")
158     in
159     let rec build_expr builder e =
160       match e with
161       | IntMatLit m -> build_matrix (fun el -> L.const_int i32_t el)
              builder m
162       | GraphLit m -> build_graph_matrix builder m
163       | FloatMatLit m -> build_matrix (fun el -> L.const_float float_t
              el) builder m
164       | IdAssign (v, e) ->
165         let comp_e = build_expr builder e in
166         (match v with
167         | s -> ignore (L.build_store comp_e (lookup s) builder));
168         comp_e
169       | Call (fname, exprs) ->
170         (match fname with
171         | "print" ->
172           (match exprs with
173           | [ e ] ->
174               build_call "matrix_print" [| build_expr builder e |] builder
175           | _ -> raise (Failure "Invalid list of expressions passed to
                  print"))
176         | "toString" ->
177           (match exprs with
178           | [ e ] ->
179             build_call
180               "matrix_tostring"
181               [| build_expr builder e |]
182               builder
183           | _ -> raise (Failure "Invalid list of expressions passed to
                  toString"))
184         | "I" ->
185           (match exprs with
186           | [ e ] ->
187             build_call
188               "matrix_create_identity"
189               [| build_expr builder e |]
190               builder
191           | _ -> raise (Failure "Invalid list of expressions passed to
                  I"))
192         | "Zero" ->
```

```
193           (match exprs with
194           | [ e ] ->
195             build_call "matrix_create_zero" [| build_expr builder e |]
                    builder
196           | _ -> raise (Failure "Invalid list of expressions passed to
                  Zero"))
197         | "range" ->
198           (match exprs with
199           | [ e ] ->
200             build_call "matrix_create_range" [| build_expr builder e |]
                    builder
201           | _ -> raise (Failure "Invalid list of expressions passed to
                  range"))
202         | "__ring_push" ->
203           (match exprs with
204           | [] -> L.build_call ring_push_f [||] "__ring_push" builder
205           | _ -> raise (Failure "Invalid list of expressions passed to
                  __ring_push"))
206         | "__ring_pop" ->
207           (match exprs with
208           | [] -> L.build_call ring_pop_f [||] "__ring_pop" builder
209           | _ -> raise (Failure "Invalid list of expressions passed to
                  __ring_pop"))
210         | f ->
211           let fdef, fdecl =
212             try StringMap.find f function_decls with
213             | Not_found ->
214               raise (Failure ("Undeclared function, " ^ f ^ ", found in
                      code generation"))
215           in
216           let args = List.map (build_expr builder) (List.rev exprs) in
217           L.build_call fdef (Array.of_list args) (fdecl.fname ^
                  "_result") builder)
218       | Binop (e1, op, e2) ->
219         let e1' = build_expr builder e1
220         and e2' = build_expr builder e2 in
221         (match op with
222         | A.Matmul -> build_call "matrix_mul" [| e1'; e2' |] builder
223         | A.Exponent -> L.build_call matrix_exp_f [| e1'; e2' |]
                "matrix_mul" builder
224         | A.Conv -> build_call "matrix_conv" [| e1'; e2' |] builder
225         | A.Elmul -> build_call "matrix_elmul" [| e1'; e2' |] builder
226         | A.Add -> build_call "matrix_eladd" [| e1'; e2' |] builder
227         | A.Concat -> build_call "matrix_concat" [| e1'; e2' |] builder
228         | A.Equal -> build_call "matrix_eq" [| e1'; e2' |] builder
229         | A.Neq -> build_call "matrix_neq" [| e1'; e2' |] builder
230         | A.Leq -> build_call "matrix_leq" [| e1'; e2' |] builder
231         | A.Less -> build_call "matrix_less" [| e1'; e2' |] builder
232         | A.Geq -> build_call "matrix_geq" [| e1'; e2' |] builder
233         | A.Greater ->
```

```
234           build_call "matrix_greater" [| e1'; e2' |] builder)
235       | UnkMatLit _ -> raise (Failure "Type of matrix is unknown")
236       | Assign _ -> raise (Failure "Assign in codegen")
237       | StringLit _ -> raise (Failure "StringLit in codegen")
238       | Unop (op, e) ->
239         let e' = build_expr builder e in
240         (match op with
241         | A.Size -> build_call "matrix_size" [| e' |] builder
242         | A.Transp -> build_call "matrix_transpose" [| e' |] builder
243         | A.Plusreduce ->
244           build_call
245             "matrix_reduce"
246             [| e'; L.const_int i32_t 0 |]
247             builder
248         | A.Mulreduce ->
249           build_call
250             "matrix_reduce"
251             [| e'; L.const_int i32_t 1 |]
252             builder
253         | A.Neg -> build_call "matrix_negate" [| e' |] builder)
254       | Id v -> L.build_load (lookup v) v builder
255       | Selection (e, args) ->
256         let partialargs' = List.map (build_expr builder) args in
257         let filledargs' = fill_select_args builder partialargs' in
258         let revfilledargs' = List.rev filledargs' in
259         let e' = build_expr builder e in
260         let args' = e' :: revfilledargs' in
261         L.build_call matrix_extract_f (Array.of_list args')
              "matrix_extract" builder
262       | SelectAssign (v, args, e) ->
263         let partialargs' = List.map (build_expr builder) args in
264         let filledargs' = fill_select_args builder partialargs' in
265         let revfilledargs' = List.rev filledargs' in
266         let e' = build_expr builder e in
267         let v' = L.build_load (lookup v) v builder in
268         let args' = v' :: e' :: revfilledargs' in
269         build_call "matrix_insert" (Array.of_list args') builder
270     in
271     let rec build_stmt builder = function
272       | Block sl -> List.fold_left build_stmt builder sl
273       | Semiring ring ->
274         ignore
275           (L.build_call
276              ring_change_f
277              [| L.const_int i32_t (List.assoc ring Definitions.rings) |]
278              "ring_change"
279              builder);
280         builder
281       | Expr e ->
282         ignore (build_expr builder e);
```

```
283         builder
284       | Return e ->
285         ignore (build_expr builder (Call ("__ring_pop", [])));
286         ignore (L.build_ret (build_expr builder e) builder);
287         builder
288       | If (pred, thn, els) ->
289         let pred_expr = build_expr builder pred in
290         let mat_truthiness =
291           L.build_call matrix_truthy_f [| pred_expr |] "matrix_truthy"
                builder
292         in
293         let bool_val =
294           L.build_icmp L.Icmp.Eq mat_truthiness (L.const_int i32_t 1)
                "i1_t" builder
295         in
296         let merge_bb = L.append_block context "merge_if" func in
297         let build_br_merge = L.build_br merge_bb in
298         let then_bb = L.append_block context "then" func in
299         add_terminal (build_stmt (L.builder_at_end context then_bb) thn)
              build_br_merge;
300         let else_bb = L.append_block context "else" func in
301         add_terminal (build_stmt (L.builder_at_end context else_bb) els)
              build_br_merge;
302         ignore (L.build_cond_br bool_val then_bb else_bb builder);
303         L.builder_at_end context merge_bb
304       | While (pred, body) ->
305         let pred_bb = L.append_block context "while" func in
306         let pred_builder = L.builder_at_end context pred_bb in
307         let pred_expr = build_expr pred_builder pred in
308         let mat_truthiness =
309           L.build_call matrix_truthy_f [| pred_expr |] "matrix_truthy"
                pred_builder
310         in
311         let bool_val =
312           L.build_icmp L.Icmp.Eq mat_truthiness (L.const_int i32_t 1)
                "i1_t" pred_builder
313         in
314         ignore (L.build_br pred_bb builder) (* builds branch to while
              from entry point *);
315         let body_bb = L.append_block context "while_body" func in
316         let body_builder = build_stmt (L.builder_at_end context body_bb)
              body in
317         add_terminal body_builder (L.build_br pred_bb);
318         let merge_bb = L.append_block context "merge" func in
319         ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
320         L.builder_at_end context merge_bb
321     in
322     let body = Expr (Call ("__ring_push", [])) :: fdecl.body in
323     let builder = build_stmt builder (Block body) in
324     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 ] }
```

```
40    ; { name = "matrix_tostring"; ret = matrix_t; args = [ matrix_t ] }
41    ; { name = "change_ring"; ret = i32_t; args = [ i32_t ] }
42    ; { name = "matrix_setelem"; ret = i32_t; args = [ matrix_t; i32_t;
          i32_t; i32_t ] }
43    ; { name = "matrix_mul"; ret = matrix_t; args = [ matrix_t; matrix_t ]
          }
44    ; { name = "matrix_conv"; ret = matrix_t; args = [ matrix_t; matrix_t
          ] }
45    ; { name = "matrix_elmul"; ret = matrix_t; args = [ matrix_t; matrix_t
          ] }
46    ; { name = "matrix_eladd"; ret = matrix_t; args = [ matrix_t; matrix_t
          ] }
47    ; { name = "matrix_extract"
48      ; ret = matrix_t
49      ; args = [ matrix_t; matrix_t; matrix_t; matrix_t; matrix_t ]
50      }
51    ; { name = "matrix_insert"
52      ; ret = matrix_t
53      ; args = [ matrix_t; matrix_t; matrix_t; matrix_t; matrix_t;
            matrix_t ]
54      }
55    ;{name = "matrix_eq"; ret = matrix_t; args = [ matrix_t; matrix_t ]}
56    ;{name = "matrix_neq"; ret = matrix_t; args = [ matrix_t; matrix_t ]}
57    ;{name = "matrix_leq"; ret = matrix_t; args = [ matrix_t; matrix_t ]}
58    ;{name = "matrix_less"; ret = matrix_t; args = [ matrix_t; matrix_t ]}
59    ;{name = "matrix_geq"; ret = matrix_t; args = [ matrix_t; matrix_t ]}
60    ;{name = "matrix_greater"; ret = matrix_t; args = [ matrix_t; matrix_t
          ]}
61    ;{name = "matrix_concat"; ret = matrix_t; args = [ matrix_t; matrix_t
          ]}
62    ;{ name = "matrix_bool"; ret = i32_t; args = [ matrix_t ] }
63    ;{ name = "matrix_negate"; ret = matrix_t; args = [ matrix_t ] }
64    ;{ name = "matrix_reduce"; ret = matrix_t; args = [ matrix_t ; i32_t] }
65    ;{ name = "matrix_insert"; ret = matrix_t; args = [ matrix_t;
          matrix_t; matrix_t; matrix_t; matrix_t; matrix_t] }
66    ;{ name = "matrix_reduce"; ret = matrix_t; args = [ matrix_t ; i32_t] }
67    ;{ name = "matrix_size"; ret = matrix_t; args = [ matrix_t ] }
68    ;{ name = "matrix_transpose"; ret = matrix_t; args = [ matrix_t ] }
69    ;{ name = "matrix_truthy"; ret = matrix_t; args = [ matrix_t ] }
70    ]
71  ;;
72
73  let matrix_truthy_t = L.function_type i32_t [| matrix_t |]
74  let matrix_truthy_f = L.declare_function "matrix_truthy" matrix_truthy_t
        blastoff_module
75
76
77  let matrix_exp_t = L.function_type matrix_t [| matrix_t; matrix_t |]
78  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
```

```
15          fprintf(stderr, "%s\n", msg);
16      exit(1);
17  }
18
19  #define GrB_die(msg, object)        \
20  do {                                \
21      const char *GrB_msg;            \
22      GrB_error(&GrB_msg, object);    \
23      fprintf(stderr, "%s\n", GrB_msg); \
24      die(msg);                       \
25  } while (0)
26
27  static int GrB_ok(GrB_Info info)
28  {
29      if (info == GrB_SUCCESS || info == GrB_NO_VALUE) {
30          return 1;
31      } else {
32          fprintf(stderr, "GrB_ok saw error code: %d\n", info);
33          return 0;
34      }
35  }
36
37  void GrB_print(GrB_Matrix mat)
38  {
39      if (!GrB_ok(GxB_Matrix_fprint(mat, NULL, GxB_COMPLETE_VERBOSE,
            stdout)))
40          die("GxB_Matrix_fprint");
41  }
42
43  void GrB_size(GrB_Matrix mat, GrB_Index *nrows, GrB_Index *ncols)
44  {
45      if (nrows && !GrB_ok(GrB_Matrix_nrows(nrows, mat)))
46          GrB_die("GrB_Matrix_nrows", mat);
47
48      if (ncols && !GrB_ok(GrB_Matrix_ncols(ncols, mat)))
49          GrB_die("GrB_Matrix_ncols", mat);
50  }
51
52  int32_t GrB_scalar(GrB_Matrix mat)
53  {
54      GrB_Index nrows, ncols;
55      int32_t elem;
56
57      GrB_size(mat, &nrows, &ncols);
58      if (nrows != 1 || ncols != 1)
59          die("GrB_scalar mat dims bad");
60
61      if (!GrB_ok(GrB_Matrix_extractElement(&elem, mat, 0, 0)))
62          GrB_die("GrB_Matrix_extractElement", mat);
63
```

```
64      return elem;
65  }
66
67  /* automatically called before main() */
68  __attribute__((constructor))
69  static void matrix_lib_init(void) {
70      if (!GrB_ok(GrB_init(GrB_NONBLOCKING)))
71          die("GrB_init");
72  }
73
74  /* automatically called after main() */
75  __attribute__((destructor))
76  void matrix_lib_finalize(void)
77  {
78      if (!GrB_ok(GrB_finalize()))
79          die("GrB_finalize");
80  }
81
82  /* BELOW: Functions used externally */
83
84  // begin ring_* functions //
85
86  // stack of rings, implemented as intrusive linked list
87  struct ring {
88      GrB_Semiring ring;
89      struct ring *prev;
90  };
91
92  struct ring *curr_ring = NULL;
93
94  void ring_push()
95  {
96      struct ring *r = malloc(sizeof(*r));
97      r->ring = GrB_PLUS_TIMES_SEMIRING_INT32;
98      r->prev = curr_ring;
99      curr_ring = r;
100 }
101
102 void ring_pop()
103 {
104     struct ring *prev;
105
106     if (!curr_ring)
107         die("ring_change: curr_ring is NULL");
108
109     prev = curr_ring->prev;
110     free(curr_ring);
111     curr_ring = prev;
112 }
113
```

```
114 void ring_change(int which)
115 {
116     if (!curr_ring)
117         die("ring_change: curr_ring is NULL");
118 
119     if (which == 0) {
120         if (!curr_ring->prev)
121             die("ring_change to #_ but curr_ring->prev is NULL");
122         curr_ring->ring = curr_ring->prev->ring;
123     } else if (which == 1) {
124         curr_ring->ring = GrB_PLUS_TIMES_SEMIRING_INT32;
125     } else if (which == 2) {
126         curr_ring->ring = GrB_LAND_LOR_SEMIRING_BOOL;
127     } else if (which == 3) {
128         curr_ring->ring = GrB_MAX_MIN_SEMIRING_INT32;
129     } else {
130         die("ring_change: unknown semiring");
131     }
132 }
133 
134 // end ring_* functions //
135 
136 // begin matrix_* functions //
137 
138 int matrix_getelem(struct matrix *A, int row, int col)
139 {
140     int32_t elem = 0;
141 
142     if (!GrB_ok(GrB_Matrix_extractElement(&elem, A->mat, row, col)))
143         GrB_die("GrB_Matrix_extractElement", A->mat);
144 
145     return elem;
146 }
147 
148 void matrix_setelem(struct matrix *A, int val, int row, int col)
149 {
150     // 0 is the implicit value; storing it explicitly would waste space
151     int32_t unused;
152     if (val == 0 &&
153             GrB_Matrix_extractElement(&unused, A->mat, row, col) ==
                    GrB_NO_VALUE)
154         return;
155 
156     if (!GrB_ok(GrB_Matrix_setElement(A->mat, val, row, col)))
157         GrB_die("GrB_Matrix_setElement", A->mat);
158 }
159 
160 
161 struct matrix *matrix_create(int nrows, int ncols)
162 {
```

```
163     struct matrix *A;
164     if (!(A = malloc(sizeof *A)))
165         die("malloc failed");
166 
167     if (!GrB_ok(GrB_Matrix_new(&A->mat, GrB_INT32, nrows, ncols)))
168         GrB_die("GrB_Matrix_new", A->mat);
169 
170     return A;
171 }
172 
173 struct matrix *matrix_create_zero(struct matrix *dims)
174 {
175     GrB_Index dim_nrows, dim_ncols, nrows, ncols;
176 
177     GrB_size(dims->mat, &dim_nrows, &dim_ncols);
178     if ((dim_nrows != 1 && dim_nrows != 2) || dim_ncols != 1)
179         die("matrix_create_zero invalid dims arg");
180 
181     nrows = matrix_getelem(dims, 0, 0);
182     ncols = dim_nrows == 2 ? matrix_getelem(dims, 1, 0) : nrows;
183 
184     return matrix_create(nrows, ncols);
185 }
186 
187 struct matrix *matrix_create_identity(struct matrix *N_scalar)
188 {
189     struct matrix *A;
190     GrB_Index i, n;
191 
192     n = GrB_scalar(N_scalar->mat);
193     A = matrix_create(n, n);
194     for (i = 0; i < n; i++)
195         matrix_setelem(A, 1, i, i);
196 
197     return A;
198 }
199 
200 struct matrix *matrix_create_range(struct matrix *range)
201 {
202     struct matrix *A;
203     int32_t lo, hi;
204     GrB_Index i, range_nrows, range_ncols;
205 
206     GrB_size(range->mat, &range_nrows, &range_ncols);
207     if (range_nrows == 1 && range_ncols == 1) {
208         lo = 0;
209         hi = matrix_getelem(range, 0, 0);
210     } else if (range_nrows == 2 && range_ncols == 1) {
211         lo = matrix_getelem(range, 0, 0);
212         hi = matrix_getelem(range, 1, 0);
```

```
213     } else {
214         die("matrix_create_range invalid range arg");
215     }
216 
217     if (lo > hi)
218         return matrix_create(0, 1);
219 
220     A = matrix_create(hi - lo, 1);
221     i = 0;
222     while (lo < hi)
223         matrix_setelem(A, lo++, i++, 0);
224 
225     return A;
226 }
227 
228 struct matrix *matrix_print(struct matrix *A)
229 {
230     GrB_Index nrows, ncols, i;
231     int elem;
232 
233     GrB_size(A->mat, &nrows, &ncols);
234     if (ncols != 1)
235         die("Tried to print string with more than 1 col");
236 
237     for (i = 0; i < nrows && (elem = matrix_getelem(A, i, 0)) != 0; i++)
238         putchar(elem);
239 
240     struct matrix *R = matrix_create(0, 0);
241     return R;
242 }
243 
244 struct matrix *matrix_tostring(struct matrix *A)
245 {
246     struct matrix *B;
247     GrB_Index nrows, ncols, i, j, k;
248     char buf[1000], *b;
249 
250     GrB_size(A->mat, &nrows, &ncols);
251     B = matrix_create(nrows * (ncols + 1) * 20, 1);
252 
253     if (nrows == 0 || ncols == 0)
254         return B;
255 
256     k = 0;
257     for (i = 0; i < nrows; i++) {
258         for (j = 0; j < ncols; j++) {
259             snprintf(buf, sizeof(buf), "%d ", matrix_getelem(A, i, j));
260             for (b = buf; *b; b++)
261                 matrix_setelem(B, *b, k++, 0);
262         }
```

```
263         matrix_setelem(B, '\n', k++, 0);
264     }
265     matrix_setelem(B, 0, k, 0);
266 
267     return B;
268 }
269 
270 
271 struct matrix *matrix_mul(struct matrix *A, struct matrix *B)
272 {
273     struct matrix *C;
274     GrB_Info info;
275     GrB_Index nrows, ncols, eq1, eq2;
276 
277     GrB_size(A->mat, &nrows, &eq1);
278     GrB_size(B->mat, &eq2, &ncols);
279     if (eq1 != eq2)
280         die("matrix_mul bad dimensions");
281 
282     C = matrix_create(nrows, ncols);
283 
284     info = GrB_mxm(C->mat,
285                    GrB_NULL,
286                    GrB_NULL,
287                    curr_ring->ring,
288                    A->mat,
289                    B->mat,
290                    GrB_NULL);
291 
292     if (!GrB_ok(info))
293         GrB_die("GrB_mxm", A->mat);
294 
295     return C;
296 }
297 
298 struct matrix *matrix_exp(struct matrix *A, struct matrix *N_scalar)
299 {
300     struct matrix *B;
301     int n;
302     GrB_Index i, nrows, ncols;
303 
304     GrB_size(A->mat, &nrows, &ncols);
305     if (nrows != ncols)
306         die("matrix_exp mat not square");
307 
308     n = GrB_scalar(N_scalar->mat);
309     if (n < 1)
310         die("matrix_exp needs positive exponent");
311 
312     B = A;
```

```
313     for (i = 0; i < n - 1; i++) {
314         B = matrix_mul(A, B);
315     }
316 
317     return B;
318 }
319 
320 struct matrix *matrix_elmul(struct matrix *A, struct matrix *B)
321 {
322     struct matrix *C;
323     GrB_Info info;
324     GrB_Index A_nrows, A_ncols, B_nrows, B_ncols;
325 
326     GrB_size(A->mat, &A_nrows, &A_ncols);
327     GrB_size(B->mat, &B_nrows, &B_ncols);
328 
329     if (A_nrows != B_nrows || A_ncols != B_ncols)
330         die("matrix_elmul bad dimensions");
331 
332     C = matrix_create(A_nrows, A_ncols);
333 
334     info = GrB_Matrix_eWiseMult_Semiring(C->mat,
335                                          GrB_NULL,
336                                          GrB_NULL,
337                                          curr_ring->ring,
338                                          A->mat,
339                                          B->mat,
340                                          GrB_NULL);
341 
342     if (!GrB_ok(info))
343         GrB_die("GrB_Matrix_eWiseMult_Semiring", A->mat);
344 
345     return C;
346 }
347 
348 struct matrix *matrix_eladd(struct matrix *A, struct matrix *B)
349 {
350     struct matrix *C;
351     GrB_Info info;
352     GrB_Index A_nrows, A_ncols, B_nrows, B_ncols;
353 
354     GrB_size(A->mat, &A_nrows, &A_ncols);
355     GrB_size(B->mat, &B_nrows, &B_ncols);
356 
357     if (A_nrows != B_nrows || A_ncols != B_ncols)
358         die("matrix_eladd bad dimensions");
359 
360     C = matrix_create(A_nrows, A_ncols);
361 
362     info = GrB_Matrix_eWiseAdd_Semiring(C->mat,
```

```
363                                     GrB_NULL,
364                                     GrB_NULL,
365                                     curr_ring->ring,
366                                     A->mat,
367                                     B->mat,
368                                     GrB_NULL);
369 
370     if (!GrB_ok(info))
371         GrB_die("GrB_Matrix_eWiseAdd_Semiring", A->mat);
372 
373     return C;
374 }
375 
376 struct matrix *matrix_extract(struct matrix *M, struct matrix *A, struct
        matrix *B, struct matrix *C, struct matrix *D)
377 {
378     struct matrix *R;
379     GrB_Index A_nrows, A_ncols, B_nrows, B_ncols, C_nrows, C_ncols,
            D_nrows, D_ncols;
380     int i, j, v, w;
381 
382     // verify that A, B, C, D are all integer matrices??
383 
384     //veryify that A, B are column vectors and that C, D are 1x1
385 
386     GrB_size(A->mat, &A_nrows, &A_ncols);
387     GrB_size(B->mat, &B_nrows, &B_ncols);
388     GrB_size(C->mat, &C_nrows, &C_ncols);
389     GrB_size(D->mat, &D_nrows, &D_ncols);
390 
391     if (A_ncols != 1 || B_ncols != 1 || C_nrows != 1 || C_ncols != 1 ||
            D_nrows != 1 || D_ncols != 1)
392         die("matrix_extract bad dimensions");
393 
394     int cval = matrix_getelem(C, 0, 0);
395     int dval = matrix_getelem(D, 0, 0);
396     R = matrix_create(A_nrows*cval, B_nrows*dval);
397 
398     //(A[i], B[j]) is top-left corner in form (cols, rows)
399     //(A[i]+v, B[j]+w) is what we iterate through
400     //(i*cval+v, j*dval+w) is where we store
401     for (i = 0; i < A_nrows; i++){
402       for (j = 0; j < B_nrows; j++){
403         int Ai = matrix_getelem(A, i, 0);
404         int Bj = matrix_getelem(B, j, 0);
405         for (v = 0; v < cval; v++){
406           for (w = 0; w < dval; w++){
407               matrix_setelem(R, matrix_getelem(M, Ai+v, Bj+w), i*cval+v,
                        j*dval+w);
408           }
```

```
409         }
410       }
411     }
412 
413     return R;
414 }
415 
416 struct matrix *matrix_insert(struct matrix *M, struct matrix *N, struct
        matrix *A, struct matrix *B, struct matrix *C, struct matrix *D)
417 {
418     //Syntax is like M[A,B,C,D] = N;
419     GrB_Index A_nrows, A_ncols, B_nrows, B_ncols, C_nrows, C_ncols,
            D_nrows, D_ncols, N_nrows, N_ncols;
420     int i, j, v, w;
421 
422     // verify that A, B, C, D are all integer matrices??
423 
424     //veryify that A, B are column vectors and that C, D are 1x1
425 
426     GrB_size(A->mat, &A_nrows, &A_ncols);
427     GrB_size(B->mat, &B_nrows, &B_ncols);
428     GrB_size(C->mat, &C_nrows, &C_ncols);
429     GrB_size(D->mat, &D_nrows, &D_ncols);
430     GrB_size(N->mat, &N_nrows, &N_ncols);
431 
432     if (A_ncols != 1 || B_ncols != 1 || C_ncols != 1 || C_nrows != 1 ||
            D_nrows != 1 || D_ncols != 1)
433         die("matrix_extract bad dimensions");
434 
435     int cval = matrix_getelem(C, 0, 0);
436     int dval = matrix_getelem(D, 0, 0);
437 
438     if ((N_nrows != cval) | (N_ncols != dval))
439         die("matrix_extract size mismatch");
440 
441     for (i = 0; i < A_nrows; i++){
442       for (j = 0; j < B_nrows; j++){
443         int Ai = matrix_getelem(A, i, 0);
444         int Bj = matrix_getelem(B, j, 0);
445         for (v = 0; v < cval; v++){
446           for (w = 0; w < dval; w++){
447               matrix_setelem(M, matrix_getelem(N, v, w), Ai+v, Bj+w);
448           }
449         }
450       }
451     }
452 
453     return N;
454 }
455 
```

```
456 struct matrix *matrix_size(struct matrix *A)
457 {
458     struct matrix *S;
459     GrB_Index nrows, ncols;
460     GrB_size(A->mat, &nrows, &ncols);
461 
462     S = matrix_create(2,1);
463 
464     matrix_setelem(S, nrows, 0, 0);
465     matrix_setelem(S, ncols, 1, 0);
466 
467     return S;
468 }
469 
470 struct matrix *matrix_reduce(struct matrix *A, int mult_flag)
471 {
472     struct matrix *R;
473     GrB_Index nrows;
474     GrB_size(A->mat, &nrows, NULL);
475 
476     GrB_Vector v;
477     GrB_Vector_new(&v, GrB_INT32, nrows) ;
478 
479     GrB_Monoid op;
480 
481     if(mult_flag){
482       GrB_BinaryOp mult;
483       GxB_Semiring_multiply(&mult, curr_ring->ring);
484       // TODO: Find a better way of doing mutliplicative identity
485       GrB_Monoid_new_INT32(&op, mult, 0);
486     } else {
487       GxB_Semiring_add(&op, curr_ring->ring);
488     }
489 
490     GrB_Matrix_reduce_Monoid(v, GrB_NULL, GrB_NULL, op, A->mat,
          GrB_NULL);
491 
492     R = matrix_create(nrows,1);
493     GrB_Col_assign(R->mat, GrB_NULL, GrB_NULL, v, GrB_ALL, nrows, 0,
          GrB_NULL);
494 
495     return R;
496 }
497 
498 struct matrix *matrix_transpose(struct matrix *A)
499 {
500     struct matrix *T;
501     GrB_Index nrows, ncols;
502     GrB_size(A->mat, &nrows, &ncols);
503 
```

```
504     T = matrix_create(ncols, nrows);
505     GrB_transpose(T->mat, GrB_NULL, GrB_NULL, A->mat, GrB_NULL);
506 
507     return T;
508 }
509 
510 struct matrix *matrix_negate(struct matrix *A)
511 {
512     struct matrix *R;
513     GrB_Index nrows, ncols;
514     int i,j;
515     GrB_size(A->mat, &nrows, &ncols);
516 
517     R = matrix_create(nrows, ncols);
518 
519     for (i = 0; i < nrows; i++) {
520         for (j = 0; j < ncols; j++) {
521           matrix_setelem(R, matrix_getelem(A, i, j) == 0, i, j);
522         }
523     }
524 
525     return R;
526 }
527 
528 struct matrix *matrix_conv(struct matrix *A, struct matrix *B)
529 {
530     struct matrix *C;
531     struct matrix *E;
532     struct matrix *f;
533     struct matrix *g;
534     struct matrix *h;
535     GrB_Index A_nrows, A_ncols, B_nrows, B_ncols, C_nrows, C_ncols;
536     int i, j;
537 
538     GrB_size(A->mat, &A_nrows, &A_ncols);
539     GrB_size(B->mat, &B_nrows, &B_ncols);
540 
541     if (A_nrows < B_nrows || A_ncols < B_ncols)
542         die("matrix_conv bad dimensions");
543 
544     // lots of memory leaked here!
545 
546     GrB_Index *row_indices, *col_indices;
547     if (!(row_indices = malloc(B_nrows * sizeof(int)))) die("malloc
          failed");
548     if (!(col_indices = malloc(B_ncols * sizeof(int)))) die("malloc
          failed");
549 
550     C_nrows = A_nrows - B_nrows + 1;
551     C_ncols = A_ncols - B_ncols + 1;
```

```
552     C = matrix_create(C_nrows, C_ncols);
553     E = matrix_create(B_nrows, B_ncols);
554     f = matrix_create(B_nrows, 1);
555     g = matrix_create(1, B_nrows);
556     h = matrix_create(1, 1);
557 
558     for (i = 0; i < C_nrows; i++) {
559         for (j = 0; j < C_ncols; j++) {
560           int k;
561           for (k = 0; k < B_nrows; k++) row_indices[k] = i+k;
562           for (k = 0; k < B_ncols; k++) col_indices[k] = j+k;
563           GrB_extract(E->mat, GrB_NULL, GrB_NULL, A->mat, row_indices,
                  B_nrows, col_indices, B_ncols, GrB_NULL);
564           E = matrix_elmul(E, B);
565           f = matrix_reduce(E, 0);
566           g = matrix_transpose(f);
567           h = matrix_reduce(g, 0);
568           matrix_setelem(C, matrix_getelem(h, 0, 0), i, j);
569         }
570     }
571 
572     return C;
573 }
574 
575 struct matrix *matrix_concat(struct matrix *A, struct matrix *B)
576 {
577     struct matrix *C;
578     GrB_Info info;
579     GrB_Index A_nrows, A_ncols, B_nrows, B_ncols;
580     int i;
581 
582     GrB_size(A->mat, &A_nrows, &A_ncols);
583     GrB_size(B->mat, &B_nrows, &B_ncols);
584 
585     if (A_ncols != B_ncols)
586         die("matrix_concat bad dimensions");
587 
588     GrB_Index *A_row_indices, *B_row_indices, *col_indices;
589     if (!(A_row_indices = malloc(A_nrows * sizeof(int)))) die("malloc
          failed");
590     if (!(B_row_indices = malloc(B_nrows * sizeof(int)))) die("malloc
          failed");
591     if (!(col_indices = malloc(A_ncols * sizeof(int)))) die("malloc
          failed");
592 
593     for (i = 0; i < A_nrows; i++) A_row_indices[i] = i;
594     for (i = A_nrows; i < A_nrows + B_nrows; i++) B_row_indices[i -
          A_nrows] = i;
595     for (i = 0; i < A_ncols; i++) col_indices[i] = i;
596 
```

```
597     C = matrix_create(A_nrows + B_nrows, A_ncols);
598 
599     info = GrB_assign(C->mat,
600                       GrB_NULL,
601                       GrB_NULL,
602                       A->mat,
603                       A_row_indices,
604                       A_nrows,
605                       GrB_ALL,
606                       A_ncols,
607                       GrB_NULL);
608 
609     info = GrB_assign(C->mat,
610                       GrB_NULL,
611                       GrB_NULL,
612                       B->mat,
613                       B_row_indices,
614                       B_nrows,
615                       GrB_ALL,
616                       B_ncols,
617                       GrB_NULL);
618 
619     if (!GrB_ok(info))
620         GrB_die("GrB_Matrix_eWiseAdd_Semiring", A->mat);
621 
622     return C;
623 }
624 
625 // Comparison operators
626 
627 struct matrix *matrix_elcompare(struct matrix *A, struct matrix *B, int
        op_index)
628 {
629     struct matrix *C;
630     int i, j;
631     int a, b, comp_val;
632 
633     GrB_Index nrows, ncols, nrowsB, ncolsB;
634     GrB_size(A->mat, &nrows, &ncols);
635     GrB_size(B->mat, &nrowsB, &ncolsB);
636 
637     /*
638     printf("dims of A: %d %d\n", (int) nrows, (int) ncols);
639     matrix_print(matrix_tostring(A));
640     printf("dims of B: %d %d\n", (int) nrowsB, (int) ncolsB);
641     matrix_print(matrix_tostring(B));
642     */
643 
644     C = matrix_create(1, 1);
645 
```

```
646     if (nrows != nrowsB || ncols != ncolsB)
647         die("Can't compare two matrices that are different dimensions");
648 
649     for (i = 0; i < nrows; i++) {
650         for (j = 0; j < ncols; j++) {
651             a = matrix_getelem(A, i, j);
652             b = matrix_getelem(B, i, j);
653             switch (op_index) {
654                 case 0: comp_val = a == b; break;
655                 case 1: comp_val = a != b; break;
656                 case 2: comp_val = a <= b; break;
657                 case 3: comp_val = a < b; break;
658                 case 4: comp_val = a >= b; break;
659                 case 5: comp_val = a > b; break;
660                 default: die("Unknown comparison operator");
661             }
662             if (!comp_val) return C;
663         }
664     }
665     matrix_setelem(C, 1, 0, 0);
666     return C;
667 }
668 
669 struct matrix *matrix_eq(struct matrix *A, struct matrix *B) { return
        matrix_elcompare(A, B, 0); }
670 struct matrix *matrix_neq(struct matrix *A, struct matrix *B) { return
        matrix_elcompare(A, B, 1); }
671 struct matrix *matrix_leq(struct matrix *A, struct matrix *B) { return
        matrix_elcompare(A, B, 2); }
672 struct matrix *matrix_less(struct matrix *A, struct matrix *B) { return
        matrix_elcompare(A, B, 3); }
673 struct matrix *matrix_geq(struct matrix *A, struct matrix *B) { return
        matrix_elcompare(A, B, 4); }
674 struct matrix *matrix_greater(struct matrix *A, struct matrix *B) {
        return matrix_elcompare(A, B, 5); }
675 
676 // "The truth value of an expr is equivalent to expr > 0" (Jake, 2021)
677 int matrix_truthy(struct matrix *A)
678 {
679     struct matrix *C;
680     struct matrix *B;
681     GrB_Index nrows, ncols;
682     GrB_size(A->mat, &nrows, &ncols);
683 
684     B = matrix_create(nrows, ncols);
685     C = matrix_greater(A, B);
686 
687     return matrix_getelem(C, 0, 0) > 0;
688 }
689 
```

```
690 // end matrix_* functions //
691
692 #ifdef RUN_TEST
693 int main(int argc, char **argv){
694     struct matrix *A, *B, *C;
695
696     ring_push();
697
698     A = matrix_create(2, 2);
699     B = matrix_create(2, 2);
700     // B = matrix_create(1, 1);
701     matrix_setelem(A, 2, 0, 0);
702     matrix_setelem(A, 2, 0, 1);
703     matrix_setelem(A, 2, 1, 0);
704     matrix_setelem(A, 2, 1, 1);
705     matrix_setelem(B, 2, 0, 0);
706     matrix_setelem(B, 2, 0, 1);
707     matrix_setelem(B, 2, 1, 0);
708     matrix_setelem(B, 2, 1, 1);
709     matrix_print(matrix_tostring(A));
710     matrix_print(matrix_tostring(B));
711
712     C = matrix_mul(A, B);
713     matrix_print(matrix_tostring(C));
714 }
715 #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.