

BLAStoff Language Final Report

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

Contents

1	Introduction	3
2	Tutorial	3
3	Language Reference Manual	4
3.1	Lexical Conventions	4
3.1.1	Assignment	4
3.1.1.1	Matrix Literal Definition	5
3.1.1.2	Graph Definition	5
3.1.1.3	Number Definition	6
3.1.1.4	Generator Function Definition	6
3.1.1.5	String Definition	8
3.1.2	Comments	8
3.1.3	Functions	8
3.1.4	If statements	9
3.1.5	For/While Loops	9
3.1.6	Operations	10
3.1.6.1	Selection	10
3.1.6.2	Matrix Multiplication	13
3.1.6.3	Convolution	13
3.1.6.4	Element-wise Multiplication	14
3.1.6.5	Element-wise Addition	15
3.1.6.6	Exponentiation	15
3.1.6.7	Size	16
3.1.6.8	Vertical Concatenation	17
3.1.6.9	A note on horizontal concatenation	17
3.1.6.10	Reduce Rows	17
3.1.6.11	A note on reduce columns	18
3.1.6.12	Assignment operators	18
3.1.6.13	Comparisons	19
3.1.6.14	Semiring redefinition	19
3.1.6.15	Logical Negation	21
3.1.7	Precedence	21
3.1.8	Keywords	22
3.2	More Language Details	22
3.2.1	Memory	22
3.2.2	Scope	22
3.2.3	Printing	23
3.3	Sample Code	23
3.3.1	Some Standard Library Functions	23
3.3.1.1	One	23
3.3.1.2	Horizontal Concatenation	24
3.3.1.3	Plus/Times Column Reduce	24

3.3.1.4	Sum	24
3.3.1.5	Range From Vector	25
3.3.2	Graph Algorithms	25
4	Project Plan	27
5	Architectural Design	28
6	Test Plan	28
7	Lessons Learned	28
7.1	Katon	28
7.2	Michael	29
7.3	Jake	29
7.4	Jason	29
8	Appendix	29

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

Welcome to BLASToff! It’s quite easy to get started. Let’s start by learning how to define a matrix, which is the only variable type in the language:

```
1 A = [5,6,7;  
2     -1,-2,-3;  
3     0,0,0];
```

As you can see, we define this matrix just by the elements in all the rows and columns. It can be quite tedious to define every matrix this way, but we have some easy syntax for things like defining the adjacency matrix of a graph by an easy listing of the directed edges of the graph:

```
1 G = [0->2;  
2     1->2;  
3     2->3  
4 ];
```

Certain common forms of matrices (this one will make a 10×15 matrix with all zeroes):

```
1 Z = Zero([10;15]);
```

Or just a 1×1 matrix.

```
1 b = 5;
```

We have a bunch of operators, defined all below, that you can use on these matrices. Let’s see how you would use addition, `+`:

```
1 X = Y + Z;
```

Now that we know how to use operations, let's look at how to write a function that actually uses them. Function definition is a mix of C style and Python style. We use the `def` keyword and don't require types for the arguments (as there is only one type!), but we have brackets around the function body. Functions can even be recursive:

```

1 def factorial(A){
2     if (A < 2){
3         return 1;
4     }
5     return A + -1;
6 }
```

As you can see, this function computes the factorial of the input. However, it will throw an error if `A` is not a 1×1 matrix, as then `A < 2 A + -1` will not be well-defined operations.

The final core functionality of BLAStoff to highlight is semiring changing, which can be used to redefine the behavior of operators. Everything we've seen so far has been in the arithmetic semiring, so let's see what happens when we change to the logical semiring, where `+` is logical or, and `\times` is logical and:

```

1 #logical;
2 print(toString(5+1)); // prints 1
3 print(toString(5+0)); // prints 1
4 print(toString(0+0)); // prints 0
5 print(toString(5*1)); // prints 1
6 print(toString(5*0)); // prints 0
7 print(toString(0*0)); // prints 0
```

3 Language Reference Manual

3.1 Lexical Conventions

3.1.1 Assignment

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

```

1 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.

3.1.1.1 Matrix Literal Definition

A matrix literal looks as follows:

```
1 [row;  
2   row;  
3   ...]
```

where each **row** looks as follows:

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

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

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

3.1.1.2 Graph Definition

The graph definition looks as follows:

```
1 [  
2   (edge | int);  
3   (edge | int);  
4   ...  
5 ]
```

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

```
1 int -> int
```

Here's an example:

```
1 G = {  
2   0->1;  
3   1->0;  
4   1->2;  
5   4;  
6 };
```

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 $a \rightarrow b$; 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).

3.1.1.3 Number Definition

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

```
1 num
```

using the Here's an example:

```
1 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:

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

which sets M as the matrix

$$[5]$$

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

3.1.1.4 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:

```
1 Zero(expr)
2 I(expr)
3 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:

```

1 A = Zero(4);
2 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:

```

1 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:

```

1 A = range(3);
2 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.

3.1.1.5 String Definition

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

```
1 'str'
```

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

```
1 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.

3.1.2 Comments

There are two types of comments in BLASoff. Single-line comments are denoted by `//`. Multi-line comments begin with `/*` and end with `*/`. For example:

```
1 A = 6; // I'm a comment!
2 B = 5; /* I'm a comment also but
3 ...
4 ...
5 I'm longer!*/
```

3.1.3 Functions

Functions in BLASoff are defined as follows:

```
1 def id(id, id, ...) {
2     stmt;
3     stmt;
4     ...
5 }
```

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

```
1 def foo(A, B) {  
2     return A;  
3 }
```

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

```
1 def bar1(A) {  
2     return;  
3 }  
4  
5 def bar2(A) {  
6     ;  
7 }
```

These two functions both return the equivalent of “None” in BLASoff, a 0×0 matrix.

3.1.4 If statements

If/else statements, look as follows:

```
1 if (expr) stmt ?[else stmt]
```

For example:

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

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

3.1.5 For/While Loops

For and while loops look as follows:

```
1 for (?expr ; expr ; ?expr) stmt  
2 while (expr) stmt
```

For example:

```

1 B = 0;
2 for (A = [0]; A < 5 ; A+=1) {
3     B+=1;
4 }
5
6 while (B > -1) {
7     B-=1;
8 }

```

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.

3.1.6 Operations

Operations are where BLASToff gets more interesting.

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

Semiring	operators		domain	0	1
	\oplus	\otimes			
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	$\mathcal{P}(\mathbb{Z})$	\emptyset	U

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	$\mathbf{C} \odot= \mathbf{A}(i, j)$
	$\mathbf{w} \odot= \mathbf{u}(i)$
assign	$\mathbf{C}(i, j) \odot= \mathbf{A}$
	$\mathbf{w}(i) \odot= \mathbf{u}$

This is how we implement these operators and some more:

3.1.6.1 Selection

Here is the grammar for the selection operator:

```

1 expr[expr, expr, expr];
2 expr[expr, expr]
3 expr[expr];

```

The BLAS_{toff} selection operator can be applied to any matrix and looks like one of the following three forms:

```

1 M[A, B, c, d];
2 M[A, B]
3 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 $[1]$. B is also optional and its default value is $[0]$. 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 . (BLAS_{toff} 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:

```

1 M = Zero(4);
2 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:

```

1 M = Zero(2);
2 M[1, 0] = 1;
3 N = Zero(3);
4 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:

```

1 v = [1;1;1];

```

```

2 v[1] = 2;
3 u = [1;1;1];
4 u[[0;2]] = 2;

```

This would result in:

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

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

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

```

1 v = Zero([5;1]);
2 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:

```

1 A = [1,2,3;
2      4,5,6;
3      7,8,9];
4 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:

```

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

```

```

5 v = [1;2;3;4];
6 u = v[[0;2;3]];

```

This would result in:

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

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

3.1.6.2 Matrix Multiplication

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

```

1 expr ? expr

```

where ? is the given operator.

The matrix multiplication operator * looks like the following:

```

1 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:

```

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

```

This would result in:

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

3.1.6.3 Convolution

The convolution operator ~ looks like the following:

```

1 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:

```

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

```

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

```

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

```

This would result in:

$$u = [8]$$

3.1.6.4 Element-wise Multiplication

The element-wise multiplication operator \odot looks like the following:

```

1 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:

```

1 A = [1,2;

```

```

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

```

This would result in:

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

3.1.6.5 Element-wise Addition

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

```

1  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:

```

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

```

This would result in:

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

3.1.6.6 Exponentiation

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

```

1  expr^(expr | T)

```

We can say these correspond to

```

1  A^b
2  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:

```

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

```

This would result in:

$$B = \begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$$

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:

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

This would result in:

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

3.1.6.7 Size

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

```
1 |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:

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

```
1 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:

```
1 m = |A|[0];  
2 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:

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

3.1.6.8 Vertical Concatenation

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

```
1 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:

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

This would result in:

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

3.1.6.9 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 BLAS uses column vectors more often than row vectors).

3.1.6.10 Reduce Rows

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

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

So, the two possible forms are

```

1 +%A
2 *%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:

```

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

```

This would result in:

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

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

3.1.6.11 A note on reduce columns

See 2.6.8.1.

3.1.6.12 Assignment operators

The operator `*=`, used as follows:

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

is equivalent to

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

The same is true for the other assignment operators:

```
1 A~=B;
```

```
2 A@=B;
```

```
3 A+=B;
```

```
4 A^=b;
```

```
5 A:=B;
```

3.1.6.13 Comparisons

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

```
1 A == B
```

```
2 A != B
```

```
3 A > B
```

```
4 A >= B
```

```
5 A < b
```

```
6 A <= B
```

where A and B are both $m \times n$ matrices. These operations return our version of “true,” $[1]$ 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 $==$.

3.1.6.14 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:

```
1 #logical
```

```
2 #arithmetic
```

```
3 #maxmin
```

```
4 #_
```

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:

```
1 a = 2;
2 b = 3;
3 c = 0;
4
5 #arithmetic;
6 a + b; //returns 5
7 a * b; //returns 6
8 a * c; //returns 0
9
10 #logical;
11 a + b; //returns 1: plus is now logical or; 0 is the only false value
    and 1 is the default true value
12 a * b; //returns 1 as well: times is now logical and
13 a * c; //returns 0
14
15
16 #maxmin;
17 a + b; //returns 3: plus is now maximum
18 a * b; //returns 2: times is now minimum
19 a * c; //returns 0
```

`#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:

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

This would result in:

$$C = \begin{bmatrix} 5 & 4 \\ 7 & 3 \end{bmatrix}$$

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

```
1 def add(x, y) {
2     return x + y;
3 }
4
5 a = 4;
6 b = 3;
7 #logical;
8
9 a + b; // will return 1
```

```
10 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):

```
1 def semiringAdd(x, y) {
2     #_;
3     return x + y;
4 }
5
6 a = 4;
7 b = 3;
8 #logical;
9
10 a + b; // will return 1
11 semiringAdd(a, b); // will also return 1
```

3.1.6.15 Logical Negation

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

```
1 !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:

```
1 A = [1,0;
2      0,3];
3 B = !A;
```

This would result in:

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

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

3.1.7 Precedence

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

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

3.1.8 Keywords

BLAStoff reserves the following keywords:

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

3.2 More Language Details

3.2.1 Memory

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

```

1 def f(x){
2     x += 1;
3 }
4 def g(x){
5     x[0] = 2;
6 }
7 a = 1;
8 f(a);
9 a == 1; //TRUE
10 g(a);
11 a == 2; //TRUE
12
13 b = 1;
14 c = b;
15 c += 1;
16 c == 2; //TRUE
17 b == 2; //FALSE
18 b == 1; //TRUE

```

3.2.2 Scope

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

1

```

2  a = 1;
3  {
4      b = 2 + a; // valid
5  }
6  c = b + 1; // valid
7
8  def f(x){
9      return x * (b + c); // error
10 }

```

3.2.3 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:

```

1  print('Hello World!');
2
3  OUTPUT:
4  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:

```

1  A = [1, 2;
2      3, 4];
3  print(toString(A));
4
5  OUTPUT:
6  1 2
7  3 4

```

3.3 Sample Code

3.3.1 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:

3.3.1.1 One

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

```

1  def One(size){
2      A = Zero(size);
3      m = size[0];

```

```

4     A[range(size[0]), range(size[1])] = 1;
5     return A;
6 }

```

3.3.1.2 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 concatenation and transpose:

```

1 def horizontalConcat(A, B){
2     return (A^T:B^T)^T;
3 }

```

3.3.1.3 Plus/Times Column Reduce

Column reduction follows similarly:

```

1 def plusColumnReduce(A){
2     #_;
3     return ((+%A)^T)^T;
4 }
5
6 def timesColumnReduce(A){
7     #_;
8     return ((*A)^T)^T;
9 }

```

3.3.1.4 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:

```

1 def sum(A){
2     #_;
3     return A~One(|A|);
4 }
5
6 def sum(A){
7     #_;
8     return plusColumnReduce(+%A);
9 }

```

3.3.1.5 Range From Vector

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

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

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

```

1 def rangeFromVector(v){
2     #logical;
3     vlogic = v~1;
4     #arithmetic;
5     n = plusColumnReduce(v); // the number of non-zero values
6     u = Zero(n, 1);
7     j = 0;
8     for (i = 0; i < |v|[0]; i += 1) {
9         if (v[i]) {
10             u[j] = i;
11             j++;
12         }
13     }
14 }
```

3.3.2 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 BLASoff

```

1 Input: graph, frontier, levels
2 depth ← 0
3 while nvals(frontier) > 0:
4     depth ← depth + 1
5     levels[frontier] ← depth
6     frontier<¬levels,replace> ← graphT ⊕.⊗ frontier
7     where ⊕.⊗ = ⊕.⊗(LogicalSemiring)
```

Our code for BFS looks like the following:

```

1 def BFS(G, frontier){
2     #logical;
3     N = |G|[0];
4     levels = Zero(N, 1);
5     maskedGT = G~T;
6     depth = 0;
7     while (sum(frontier)) {
```

```

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

```

Let's look at how this code works. 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

Table 1: Team Roles

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

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!

4 Project Plan

Workflow:

We used GitHub 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.

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

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 Ocamlfmtr with the Jane Street profile to unify the code standard.

5 Architectural Design

6 Test Plan

7 Lessons Learned

7.1 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.

7.2 Michael

7.3 Jake

7.4 Jason

8 Appendix

```
1 (* Top-level of the BLASToff compiler: scan & parse the input,
2    check the resulting AST and generate an SAST from it, generate LLVM
3    IR,
4    and dump the module *)
5
6 type action =
7   | Ast
8   | Semant
9   | LLVM_IR
10  | Compile
11
12 let () =
13   let action = ref Compile in
14   let set_action a () = action := a in
15   let speclist =
16     [ "-a", Arg.Unit (set_action Ast), "Print the AST"
17     ; "-s", Arg.Unit (set_action Semant), "Print the SAST"
18     ; "-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR"
19     ; ( "-c"
20       , Arg.Unit (set_action Compile)
21       , "Check and print the generated LLVM IR (default)" )
22     ]
23   in
24   let usage_msg = "usage: ./blastoff.native [-a|-s|-l|-c] [file.blst]" in
25   let channel = ref stdin in
26   Arg.parse speclist (fun filename -> channel := open_in filename)
27     usage_msg;
28   let lexbuf = Lexing.from_channel !channel in
29   let scanner_token_wrapper lb =
30     let tok = Scanner.token lb in
31     tok
32   in
33   let ast = Blastoffparser.program scanner_token_wrapper lexbuf in
34   match !action with
35   | Ast -> print_string (Ast.string_of_program ast)
36   | _ ->
37     let sast =
```

```

36     try Semant.check ast with
37     | e ->
38         let msg = Printexc.to_string e in
39         raise (Failure ("Semantic Checking Error: " ^ msg))
40     in
41     (match !action with
42     | Ast -> ()
43     | Semant -> print_string (Ast.string_of_program sast)
44     | LLVM_IR -> print_string (Llvm.string_of_llmodule
45                               (Codegen.translate sast))
46     | Compile ->
47         let m = Codegen.translate sast in
48         Llvm_analysis.assert_valid_module m;
49         print_string (Llvm.string_of_llmodule m))
50 ;;

```

```

1  (* Ocamllex scanner for BLASToff *)
2
3  { open Blastoffparser
4
5  (* http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual026.html#toc111
6  *)
7  let keyword_table = Hashtbl.create 97
8  let _ = List.iter (fun (kwd, tok) -> Hashtbl.add keyword_table kwd tok)
9      [ "while", WHILE;
10        "return", RETURN;
11        "if", IF;
12        "else", ELSE;
13        "for", FOR;
14        "def", FDECL;
15        "T", TRANSP]
16
17  }
18
19  let digit = ['0'-'9']
20  let arrow = ['-'] ['>']
21
22  rule token = parse
23      [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
24      | "/" { comment lexbuf } (* Comments *)
25      | "/" { single_line_comment lexbuf }
26      | '-'?digit* as lxm { INTLITERAL(int_of_string lxm) }
27      | ['-']?digit*['.']*digit* as lxm { FLOATLITERAL(float_of_string lxm) }
28      | '|' { VLINE }
29      | '[' { LBRACK }
30      | ']' { RBRACK }
31      | '(' { LPAREN }
32      | ')' { RPAREN }
33      | '{' { LBRACE }
34      | '}' { RBRACE }

```

```

33 | '\''[^\''']*'\'' as str { STRINGLITERAL(String.sub str 1
    | ((String.length str) - 2)) }
34 | '@' { ELMUL }
35 | "@=" { ELMULASSIGN }
36 | '~' { CONV }
37 | "~=" { CONVASSIGN }
38 | ':' { CONCAT }
39 | ":=" { CONCATASSIGN }
40 | ';' { SEMI }
41 | ',' { COMMA }
42 | '+' { PLUS }
43 | "+=" { PLUSASSIGN }
44 | '*' { MATMUL }
45 | "*=" { MATMULASSIGN }
46 | '=' { ASSIGN }
47 | arrow { EDGE }
48 | ['+']['%'] { PLUSREDUCE }
49 | ['*']['%'] { MULREDUCE }
50 | "==" { EQ }
51 | "!=" { NEQ }
52 | '<' { LT }
53 | "<=" { LEQ }
54 | ">" { GT }
55 | ">=" { GEQ }
56 | '^' { RAISE }
57 | "^=" { RAISEASSIGN }
58 | '!' { NOT }
59 | '#' { SEMIRING }
60 | ['a'-'z' 'A'-'Z' '_']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm
61 | { (*print_endline "find lxm: ";
62 |   print_endline lxm;*)
63 |   try
64 |     Hashtbl.find keyword_table lxm
65 |   with Not_found ->
66 |     ID(lxm)}
67 | eof { EOF }
68 | _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }
69
70 and comment = parse
71   "*/" { token lexbuf }
72 | _ { comment lexbuf }
73 and single_line_comment = parse
74   '\n' { token lexbuf }
75 | _ { single_line_comment lexbuf }

```

```

1 (* Abstract Syntax Tree and functions for printing it *)
2
3 type op =
4   | Add

```



```

5   | Matmul
6   | Elmul
7   | Conv
8   | Equal
9   | Neq
10  | Less
11  | Leq
12  | Greater
13  | Geq
14  | Concat
15  | Exponent
16
17  type uop =
18    | Neg
19    | Transp
20    | Plusreduce
21    | Mulreduce
22    | Size
23
24  type lit =
25    | IntLit of int
26    | FloatLit of float
27
28  type expr =
29    | GraphLit of (int * int) list
30    | UnkMatLit of lit list list
31    | IntMatLit of int list list
32    | FloatMatLit of float list list
33    | Id of string
34    | Binop of expr * op * expr
35    | 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
79         :: acc) [] row)
80     in
81     "["
82     ^ String.concat ";" (List.fold_left (fun acc row -> string_of_row row
83       :: acc) [] m)
84     ^ "]"
85   ;;
86
87   let string_of_graph g =
88     let string_of_edge (v1, v2) = string_of_int v1 ^ "->" ^ string_of_int
89       v2 in
90     "[" ^ String.concat ";" (List.map string_of_edge g) ^ "]"
91   ;;
92
93   let rec string_of_expr = function
94   | Id s -> s
95   | Binop (e1, o, e2) ->
96     string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
97   | Unop (o, e) -> string_of_e_with_uop e o
98   | Assign (e1, e2) -> string_of_expr e1 ^ " = " ^ string_of_expr e2
99   | IdAssign (s, e) -> s ^ " = " ^ string_of_expr e
100  | Call (f, el) -> f ^ "(" ^ String.concat "," (List.map
    string_of_expr el) ^ ")"
101  | UnkMatLit m ->
    string_of_mat
    (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
111         args) ^ "]"
112 | SelectAssign (s, args, e) ->
113     s
114     ^ "["
115     ^ String.concat ", " (List.map string_of_expr args)
116     ^ "]"
117     ^ " = "
118     ^ string_of_expr e
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"
131 | Block stmts -> "{\n" ^ String.concat "" (List.map string_of_stmt
132     stmts) ^ "}\n"
133 | Expr expr -> string_of_expr expr ^ ";\n"
134 | Return expr -> "return " ^ string_of_expr expr ^ ";\n"
135 | If (e, s, Block []) -> "if (" ^ string_of_expr e ^ ")\n" ^
136     string_of_stmt s
137 | If (e, s1, s2) ->
138     "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s1 ^ "else\n" ^
139     string_of_stmt s2
140 | While (e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt
141     s
142 ;;
143
144 let string_of_func func =
145     "def "
146     ^ func.fname
147     ^ "("
148     ^ String.concat ", " func.formals
149     ^ ")"

```

```

146   ~ "{\n"
147   ~ String.concat "" (List.map string_of_stmt func.body)
148   ~ "}\n"
149   ;;
150
151   let string_of_program (funcs, stmts) =
152     String.concat "" (List.map string_of_func funcs)
153     ~ "\n"
154     ~ String.concat "" (List.map string_of_stmt stmts)
155   ;;

```

```

1  /* Ocaml yacc parser for BLASToff */
2
3  %{
4  open Ast
5  %}
6
7  %token SEMI LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK COMMA SEMIRING EDGE
8  %token MATMUL ELMUL ASSIGN FDECL RANGEMAT CONV PLUS RAISE PLUSREDUCE
9  %token NOT EQ NEQ LT LEQ GT GEQ IMAT ELMAT TRANSP VLINE SEMIRING CONCAT
10 %token RETURN IF ELSE FOR WHILE INT BOOL FLOAT VOID
11 %token PLUSASSIGN ELMULASSIGN CONVASSIGN MATMULASSIGN CONCATASSIGN
12 %token <int> INTLITERAL
13 %token <float> FLOATLITERAL
14 %token <string> STRINGLITERAL
15 %token <string> ID
16 %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
24 %right CONCATASSIGN RAISEASSIGN
25 %left EQ NEQ
26 %left LT GT LEQ GEQ
27 %right LBRACK RBRACK
28 %left PLUS
29 %left MATMUL ELMUL
30 %left CONCAT CONV
31 %left RAISE
32 %left EDGE
33 %right PLUSREDUCE MULREDUCE
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,
76     Block([$8 ; Expr($6)])))]}

```

```

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)) }
108    | expr RAISE TRANSP { Unop(Transp, $1) }
109    | NOT expr { Unop(Neg, $2) }
110    | PLUSREDUCE expr { Unop(Plusreduce, $2) }
111    | MULREDUCE expr { Unop(Mulreduce, $2) }
112    | expr LBRACK expr_list RBRACK { Selection($1, $3) }
113    | expr ASSIGN expr { Assign($1, $3) }
114    | ID LPAREN args_opt RPAREN { Call($1, $3) }
115    | LPAREN expr RPAREN { $2 }
116    | VLINE expr VLINE { Unop(Size, $2) }
117    | LBRACK mat_content RBRACK { UnkMatLit($2) }
118    | LBRACK graph_content RBRACK { GraphLit($2) }
119
120 mat_content:
121     mat_row { [$1] }
122     | mat_content SEMI mat_row {$3 :: $1}
123
124 mat_row:
125     lit { [$1] }
126     | mat_row COMMA lit {$3 :: $1 }

```

```

127 | /* nothing */ {[]}
128
129 graph_content:
130   edge { [$1] }
131   | graph_content SEMI edge {$3 :: $1}
132
133 edge:
134   INTLITERAL EDGE INTLITERAL { ($1, $3) }
135
136 args_opt:
137   /* nothing */ { [] }
138   | args_list { List.rev $1 }
139
140 args_list:
141   expr { [$1] }
142   | args_list COMMA expr { $3 :: $1 }

```

```

1 (* Semantic checking for the BLASToff compiler *)
2
3 open Ast
4 module StringMap = Map.Make (String)
5
6 (* Semantic checking of the AST. Returns an SAST if successful,
7  throws an exception if something is wrong.
8  Check each global variable, then check each function *)
9
10 let check (funcs, stmts) =
11   let check_vars loc stmt_lst =
12     let add_decl lst = function
13       | 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
23         ^ " in " ^ loc))
24       | _ :: tl -> check_dups tl
25     in
26     check_dups (List.sort compare decls)
27   in
28   (**** Check functions ****)
29
30   (* Collect function declarations for built-in functions: no bodies *)
31   let built_in_decls =
32     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
59     lst) m in
60   let get_char_codes s =
61     (* Takes string, returns backwards list of character codes *)
62     let rec exp i l = if i < 0 then l else exp (i - 1) (Char.code s.[i]
63       :: l) in
64     exp (String.length s - 1) []
65   in
66   let rec check_expr = function
67     | Call (fname, args) as call ->
68       let fd = find_func fname in
69       let num_formals = List.length fd.formals in
70       if List.length args != num_formals
71       then
72         raise
73         (Failure
74          ("Expecting "
75           ^ string_of_int num_formals
76           ^ " arguments in "
77           ^ string_of_expr call))
78       else Call (fname, List.map check_expr args)
79     | StringLit s ->
80       let chars = List.rev (get_char_codes s) in
81       IntMatLit (List.map (fun c -> [ c ]) chars)

```



```

80 | UnkMatLit m ->
81 | let has_float = contains_float m in
82 | (match has_float with
83 | true ->
84 |   FloatMatLit
85 |   (List.map
86 |   (fun row ->
87 |     List.map
88 |     (function
89 |       | IntLit lit -> float_of_int lit
90 |       | FloatLit lit -> lit)
91 |     row
92 |   ) m)
93 | false ->
94 |   IntMatLit
95 |   (List.map
96 |   (fun row ->
97 |     List.map
98 |     (function
99 |       | IntLit lit -> lit
100 |      | FloatLit _ -> raise (Failure "Expected Integers in
101 |                                Matrix"))
102 |     row)
103 |   m))
104 | Id n -> Id n
105 | Binop (e1, op, e2) -> Binop (check_expr e1, op, check_expr e2)
106 | Unop (op, e) -> Unop (op, check_expr e)
107 | FloatMatLit _ -> raise (Failure "Unexpected float matrix in semant
108 |                                checking")
109 | IntMatLit _ -> raise (Failure "Unexpected float matrix in semant
110 |                                checking")
111 | GraphLit g -> GraphLit g
112 | Selection (e, args) -> Selection (check_expr e, List.map
113 |   check_expr args)
114 | IdAssign (n, e) -> IdAssign (n, check_expr e)
115 | SelectAssign (n, args, e) -> SelectAssign (n, List.map check_expr
116 |   args, check_expr e)
117 | Assign (e1, e2) ->
118 | let fix_assign = function
119 |   | Id i, e -> check_expr (IdAssign (i, e))
120 |   | Selection (Id n, args), e -> check_expr (SelectAssign (n,
121 |     args, e))
122 |   | _ -> raise (Failure "Bad left side of assignment, expected ID
123 |                         or ID[...]")
124 | in
125 | fix_assign (e1, e2)
126 | in
127 | let rec check_stmt = function
128 |   | Expr e -> Expr (check_expr e)
129 |   | Semiring ring ->

```

```

123     (match List.mem_assoc ring Definitions.rings with
124     | true -> Semiring ring
125     | false -> raise (Failure ("Unknown semiring " ^ ring)))
126   | Block b1 -> Block (check_stmt_list b1)
127   | If (p, b1, b2) -> If (check_expr p, check_stmt b1, check_stmt b2)
128   | While (p, s) -> While (check_expr p, check_stmt s)
129   | Return e -> Return (check_expr e)
130 and check_stmt_list = function
131   | [ (Return _ as s) ] -> [ check_stmt s ]
132   | Return _ :: _ -> raise (Failure "Unreachable statments after
133     return")
134   | Block s1 :: ss -> check_stmt_list (s1 @ ss)
135   | s :: ss -> check_stmt s :: check_stmt_list ss
136   | [] -> []
137 in
138 let add_return body =
139   match List.rev body with
140   | Return _ :: _ -> body
141   | _ as l -> List.rev (Return (UnkMatLit [ [] ]) :: l)
142 in
143 let check_function func =
144   let _ = check_vars "body" func.body in
145   let checked_body = check_stmt_list (add_return func.body) in
146   { fname = func.fname; formals = func.formals; body = checked_body }
147 in
148 List.map check_function funcs, List.map check_stmt stmts
149 ;;

```

```

1 module A = Ast
2 open Ast
3 open Definitions
4 module StringMap = Map.Make (String)
5
6 let translate (functions, statements) =
7   let main_fdecl = { fname = "main"; formals = []; body = List.rev
8     statements } in
9   let function_decls : (L.lvalue * func_decl) StringMap.t =
10     let function_decl m fdecl =
11       let name = fdecl.fname
12       and formal_types = Array.of_list (List.map (fun _ -> matrix_t)
13         fdecl.formals) in
14       let ftype = L.function_type matrix_t formal_types in
15       StringMap.add name (L.define_function name ftype blastoff_module,
16         fdecl) m
17   in
18   let decls = List.fold_left function_decl StringMap.empty functions in
19   StringMap.add
20     main_fdecl.fname
21     ( L.define_function

```

```

19     main_fdecl.fname
20     (L.function_type i32_t (Array.of_list []))
21     blastoff_module
22     , main_fdecl )
23     decls
24 in
25 let build_function_body fdecl is_main =
26     let func, _ =
27         try StringMap.find fdecl.fname function_decls with
28         | Not_found -> raise (Failure ("Unknown function, " ^ fdecl.fname))
29     in
30     let builder = L.builder_at_end context (L.entry_block func) in
31     let local_vars =
32         let add_formal m n p =
33             L.set_value_name n p;
34             let local = L.build_alloca matrix_t n builder in
35             ignore (L.build_store p local builder);
36             StringMap.add n local m
37         in
38         let add_local m n =
39             if StringMap.mem n m
40             then m
41             else (
42                 let local_var = L.build_alloca matrix_t n builder in
43                 StringMap.add n local_var m)
44         in
45         let formals =
46             List.fold_left2
47                 add_formal
48                 StringMap.empty
49                 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)
79     (snd elem)) 0 m in
80   let mat =
81     L.build_call
82       matrix_create_f
83       [| L.const_int i32_t dim; L.const_int i32_t dim |]
84       "matrix_create"
85       builder
86   in
87   List.iter
88     (fun elem ->
89       ignore
90         (L.build_call
91           matrix_setelem_f
92           [| mat
93             ; L.const_int i32_t 1
94             ; L.const_int i32_t (fst elem)
95             ; L.const_int i32_t (snd elem)
96             |]
97           "matrix_setelem"
98           builder))
99     m;
100   mat
101 in
102 let build_matrix typ builder m =
103   let mat =
104     L.build_call
105       matrix_create_f
106       [| L.const_int i32_t (List.length m)
107         ; L.const_int i32_t (List.length (List.hd m))
108         |]
109       "matrix_create"
110       builder
111   in
112   List.iteri
113     (fun i row ->
114       (List.iteri (fun j elem ->
115         ignore
116           (L.build_call
117             matrix_setelem_f
118             [| mat

```

```

118             ; typ elem
119             ; L.const_int i32_t i
120             ; L.const_int i32_t j
121             []
122             "matrix_setelem"
123             builder)))
124         (List.rev row))
125     (List.rev m);
126     mat
127 in
128 let rec fill_select_args builder args =
129     let zero =
130         L.build_call
131             matrix_create_f
132             [| L.const_int i32_t 1; L.const_int i32_t 1 |]
133             "matrix_create"
134             builder
135     in
136     let base =
137         L.build_call
138             matrix_create_f
139             [| L.const_int i32_t 1; L.const_int i32_t 1 |]
140             "matrix_create"
141             builder
142     in
143     let one =
144         ignore
145         (L.build_call
146             matrix_setelem_f
147             [| base; L.const_int i32_t 1; L.const_int i32_t 0;
148                 L.const_int i32_t 0 |]
149             "matrix_setelem"
150             builder);
151     base
152 in
153 match args with
154 | [ _; _; _; _ ] as l -> 1
155 | [ _; _; _ ] as l -> fill_select_args builder (one :: l)
156 | [ _; _ ] as l -> fill_select_args builder (one :: l)
157 | [ _ ] as l -> fill_select_args builder (zero :: l)
158 | _ -> raise (Failure "Too many/few arguments to selection")
159 in
160 let rec build_expr builder e =
161     match e with
162     | IntMatLit m -> build_matrix (fun el -> L.const_int i32_t el)
163         builder m
164     | GraphLit m -> build_graph_matrix builder m
165     | FloatMatLit m -> build_matrix (fun el -> L.const_float float_t
166         el) builder m
167     | 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
176                               print"))
177     | "toString" ->
178         (match exprs with
179         | [ e ] ->
180             build_call
181                 "matrix_tostring"
182                 [| build_expr builder e |]
183                 builder
184         | _ -> raise (Failure "Invalid list of expressions passed to
185                               toString"))
186     | "I" ->
187         (match exprs with
188         | [ e ] ->
189             build_call
190                 "matrix_create_identity"
191                 [| build_expr builder e |]
192                 builder
193         | _ -> raise (Failure "Invalid list of expressions passed to
194                               I"))
195     | "Zero" ->
196         (match exprs with
197         | [ e ] ->
198             build_call "matrix_create_zero" [| build_expr builder e |]
199             builder
200         | _ -> raise (Failure "Invalid list of expressions passed to
201                               Zero"))
202     | "range" ->
203         (match exprs with
204         | [ e ] ->
205             build_call "matrix_create_range" [| build_expr builder e |]
206             builder
207         | _ -> raise (Failure "Invalid list of expressions passed to
208                               range"))
209     | "__ring_push" ->
210         (match exprs with
211         | [] -> L.build_call ring_push_f [||] "__ring_push" builder
212         | _ -> raise (Failure "Invalid list of expressions passed to
213                               __ring_push"))
214     | "__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_exp" 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 ->
                build_call "matrix_greater" [| e1'; e2' |] builder)
234     | UnkMatLit _ -> raise (Failure "Type of matrix is unknown")
235     | Assign _ -> raise (Failure "Assign in codegen")
236     | StringLit _ -> raise (Failure "StringLit in codegen")
237     | Unop (op, e) ->
238         let e' = build_expr builder e in
239         (match op with
240         | A.Size -> build_call "matrix_size" [| e' |] builder
241         | A.Transp -> build_call "matrix_transpose" [| e' |] builder
242         | A.Plusreduce ->
                build_call
243                 "matrix_reduce"
244                 [| e'; L.const_int i32_t 0 |]
                builder
245         | A.Mulreduce ->
                build_call
246                 "matrix_reduce"
247                 [| e'; L.const_int i32_t 1 |]
                builder
248         )
249     )
250
251
252

```

```

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
325     builder
326     (L.build_ret (L.const_int (if is_main then i32_t else matrix_t) 0))
327 in
328 build_function_body main_fdecl true;
329 List.iter2 build_function_body functions (List.map (fun _ -> false)
    functions);
330 blastoff_module
331 ;;

```

```

1 module L = Llvm
2
3 let context = L.global_context ()
4 let llmem = L.MemoryBuffer.of_file "graphblas.bc"
5 let llm = Llvm_bitreader.parse_bitcode context llmem
6 let blastoff_module = L.create_module context "BLAStoff"
7 let rings = [ "_", 0; "arithmetic", 1; "logical", 2; "maxmin", 3 ]
8
9 let functions =

```

```

10   [ "I", [ "n" ]
11   ; "Zero", [ "d" ]
12   ; "range", [ "n" ]
13   ; "print", [ "e" ]
14   ; "toString", [ "e" ]
15   ]
16   ;;
17
18   type built_in =
19   { name : string
20   ; ret : L.lltype
21   ; args : L.lltype list
22   }
23
24   let i32_t = L.i32_type context
25   let float_t = L.double_type context
26
27   let matrix_t =
28   L.pointer_type
29   (match L.type_by_name llm "struct.matrix" with
30   | None -> raise (Failure "matrix type implementation not found")
31   | Some t -> t)
32   ;;
33
34   let built_in_defs : built_in list =
35   [ { name = "matrix_create"; ret = matrix_t; args = [ matrix_t ] }
36   ; { name = "matrix_create_identity"; ret = matrix_t; args = [ matrix_t
37   ] }
38   ; { name = "matrix_create_zero"; ret = matrix_t; args = [ matrix_t ] }
39   ; { name = "matrix_create_range"; ret = matrix_t; args = [ matrix_t ] }
40   ; { name = "matrix_print"; ret = matrix_t; args = [ matrix_t ] }
41   ; { name = "matrix_tostring"; ret = matrix_t; args = [ matrix_t ] }
42   ; { name = "change_ring"; ret = i32_t; args = [ i32_t ] }
43   ; { name = "matrix_setelem"; ret = i32_t; args = [ matrix_t; i32_t;
44   i32_t; i32_t ] }
45   ; { name = "matrix_mul"; ret = matrix_t; args = [ matrix_t; matrix_t ]
46   }
47   ; { name = "matrix_conv"; ret = matrix_t; args = [ matrix_t; matrix_t
48   ] }
49   ; { name = "matrix_elmul"; ret = matrix_t; args = [ matrix_t; matrix_t
50   ] }
51   ; { name = "matrix_eladd"; ret = matrix_t; args = [ matrix_t; matrix_t
52   ] }
53   ; { name = "matrix_extract"
54   ; ret = matrix_t
55   ; args = [ matrix_t; matrix_t; matrix_t; matrix_t; matrix_t ]
56   }
57   ; { name = "matrix_insert"
58   ; ret = matrix_t
59   ; 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
79
80 let create_fun_type fdef = L.function_type fdef.ret (Array.of_list
    fdef.args)
81 let declare_fun fname ftype = L.declare_function fname ftype
    blastoff_module
82 let built_ins = List.map (fun fdef -> fdef.name, declare_fun fdef.name
    (create_fun_type fdef)) built_in_defs
83 let build_call fname args builder = L.build_call (List.assoc fname
    built_ins) args fname builder
84
85 let matrix_create_t = L.function_type matrix_t [| i32_t; i32_t |]
86 let matrix_create_f = L.declare_function "matrix_create" matrix_create_t
    blastoff_module
87 let matrix_identity_t = L.function_type matrix_t [| matrix_t |]
88 let matrix_identity_f =
89     L.declare_function "matrix_create_identity" matrix_identity_t
    blastoff_module
90 let ring_push_t = L.function_type i32_t [| |]
91 let ring_push_f = L.declare_function "ring_push" ring_push_t

```

```

        blastoff_module
92 let ring_pop_t = L.function_type i32_t [|]
93 let ring_pop_f = L.declare_function "ring_pop" ring_pop_t blastoff_module
94 let ring_change_t = L.function_type i32_t [| i32_t |]
95 let ring_change_f = L.declare_function "ring_change" ring_change_t
        blastoff_module
96 let matrix_setelem_t = L.function_type i32_t [| matrix_t; i32_t; i32_t;
        i32_t |]
97 let matrix_setelem_f =
98     L.declare_function "matrix_setelem" matrix_setelem_t blastoff_module
99 let matrix_extract_t =
100     L.function_type matrix_t [| matrix_t; matrix_t; matrix_t; matrix_t;
        matrix_t |]
101 let matrix_extract_f =
102     L.declare_function "matrix_extract" matrix_extract_t blastoff_module

```

```

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <errno.h>
4 #include <GraphBLAS.h>
5
6 struct matrix {
7     GrB_Matrix mat;
8 };
9
10 static void die(const char *msg)
11 {
12     if (errno)
13         perror(msg);
14     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 }

```

```

34     }
35 }
36
37 void GrB_print(GrB_Matrix mat)
38 {
39     if (!GrB_ok(GxB_Matrix_fprint(mat, NULL, GxB_COMPLETE_VERBOSE,
40         stdout)))
41         die("GxB_Matrix_fprint");
42 }
43
44 void GrB_size(GrB_Matrix mat, GrB_Index *nrows, GrB_Index *ncols)
45 {
46     if (nrows && !GrB_ok(GrB_Matrix_nrows(nrows, mat)))
47         GrB_die("GrB_Matrix_nrows", mat);
48
49     if (ncols && !GrB_ok(GrB_Matrix_ncols(ncols, mat)))
50         GrB_die("GrB_Matrix_ncols", mat);
51 }
52
53 int32_t GrB_scalar(GrB_Matrix mat)
54 {
55     GrB_Index nrows, ncols;
56     int32_t elem;
57
58     GrB_size(mat, &nrows, &ncols);
59     if (nrows != 1 || ncols != 1)
60         die("GrB_scalar mat dims bad");
61
62     if (!GrB_ok(GrB_Matrix_extractElement(&elem, mat, 0, 0)))
63         GrB_die("GrB_Matrix_extractElement", mat);
64
65     return elem;
66 }
67
68 /* automatically called before main() */
69 __attribute__((constructor))
70 static void matrix_lib_init(void) {
71     if (!GrB_ok(GrB_init(GrB_NONBLOCKING)))
72         die("GrB_init");
73 }
74
75 /* automatically called after main() */
76 __attribute__((destructor))
77 void matrix_lib_finalize(void)
78 {
79     if (!GrB_ok(GrB_finalize()))
80         die("GrB_finalize");
81 }
82
83 /* 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) ==
154         GrB_NO_VALUE)
155         return;
156
157     if (!GrB_ok(GrB_Matrix_setElement(A->mat, val, row, col)))
158         GrB_die("GrB_Matrix_setElement", A->mat);
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     GrB_size(A->mat, &nrows, &ncols);
233     if (ncols != 1)
234         die("Tried to print string with more than 1 col");
235
236     for (i = 0; i < nrows && (elem = matrix_getelem(A, i, 0)) != 0; i++)
237         putchar(elem);
238
239     struct matrix *R = matrix_create(0, 0);
240     return R;
241 }
242
243 struct matrix *matrix_tostring(struct matrix *A)
244 {
245     struct matrix *B;
246     GrB_Index nrows, ncols, i, j, k;
247     char buf[1000], *b;
248
249     GrB_size(A->mat, &nrows, &ncols);
250     B = matrix_create(nrows * (ncols + 1) * 20, 1);
251
252     if (nrows == 0 || ncols == 0)
253         return B;
254
255     k = 0;
256     for (i = 0; i < nrows; i++) {
257         for (j = 0; j < ncols; j++) {
258             snprintf(buf, sizeof(buf), "%d ", matrix_getelem(A, i, j));
259             for (b = buf; *b; b++)
260                 matrix_setelem(B, *b, k++, 0);
261         }
262         matrix_setelem(B, '\n', k++, 0);
263     }
264     matrix_setelem(B, 0, k, 0);
265
266     return B;
267 }
268
269
270 struct matrix *matrix_mul(struct matrix *A, struct matrix *B)
271 {
272     struct matrix *C;
273     GrB_Info info;
274     GrB_Index nrows, ncols, eq1, eq2;
275
276     GrB_size(A->mat, &nrows, &eq1);
277     GrB_size(B->mat, &eq2, &ncols);
278     if (eq1 != eq2)
279         die("matrix_mul bad dimensions");
280
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
377                               matrix *B, struct matrix *C, struct matrix *D)
378 {
379     struct matrix *R;
380     GrB_Index A_nrows, A_ncols, B_nrows, B_ncols, C_nrows, C_ncols,
381               D_nrows, D_ncols;

```

```

380     int i, j, v, w;
381
382     // verify that A, B, C, D are all integer matrices??
383
384     //verify 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 ||
392         D_nrows != 1 || D_ncols != 1)
393         die("matrix_extract bad dimensions");
394
395     int cval = matrix_getelem(C, 0, 0);
396     int dval = matrix_getelem(D, 0, 0);
397     R = matrix_create(A_nrows*cval, B_nrows*dval);
398
399     //(A[i], B[j]) is top-left corner in form (cols, rows)
400     //(A[i]+v, B[j]+w) is what we iterate through
401     //(i*cval+v, j*dval+w) is where we store
402     for (i = 0; i < A_nrows; i++){
403         for (j = 0; j < B_nrows; j++){
404             int Ai = matrix_getelem(A, i, 0);
405             int Bj = matrix_getelem(B, j, 0);
406             for (v = 0; v < cval; v++){
407                 for (w = 0; w < dval; w++){
408                     matrix_setelem(R, matrix_getelem(M, Ai+v, Bj+w), i*cval+v,
409                                     j*dval+w);
410                 }
411             }
412         }
413     }
414     return R;
415 }
416
417 struct matrix *matrix_insert(struct matrix *M, struct matrix *N, struct
418                             matrix *A, struct matrix *B, struct matrix *C, struct matrix *D)
419 {
420     //Syntax is like M[A,B,C,D] = N;
421     GrB_Index A_nrows, A_ncols, B_nrows, B_ncols, C_nrows, C_ncols,
422         D_nrows, D_ncols, N_nrows, N_ncols;
423     int i, j, v, w;
424
425     // verify that A, B, C, D are all integer matrices??
426
427     //verify that A, B are column vectors and that C, D are 1x1

```

```

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,
491                             GrB_NULL);
492
493     R = matrix_create(nrows,1);
494     GrB_Col_assign(R->mat, GrB_NULL, GrB_NULL, v, GrB_ALL, nrows, 0,
495                  GrB_NULL);
496
497     return R;
498 }
499
500 struct matrix *matrix_transpose(struct matrix *A)
501 {
502     struct matrix *T;
503     GrB_Index nrows, ncols;
504     GrB_size(A->mat, &nrows, &ncols);
505
506     T = matrix_create(ncols, nrows);
507     GrB_transpose(T->mat, GrB_NULL, GrB_NULL, A->mat, GrB_NULL);
508
509     return T;
510 }
511
512 struct matrix *matrix_negate(struct matrix *A)
513 {
514     struct matrix *R;
515     GrB_Index nrows, ncols;
516     int i,j;
517     GrB_size(A->mat, &nrows, &ncols);
518
519     R = matrix_create(nrows, ncols);
520
521     for (i = 0; i < nrows; i++) {
522         for (j = 0; j < ncols; j++) {
523             matrix_setelem(R, matrix_getelem(A, i, j) == 0, i, j);
524         }
525     }
526 }

```

```

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
548         failed");
549     if (!(col_indices = malloc(B_ncols * sizeof(int)))) die("malloc
550         failed");
551
552     C_nrows = A_nrows - B_nrows + 1;
553     C_ncols = A_ncols - B_ncols + 1;
554     C = matrix_create(C_nrows, C_ncols);
555     E = matrix_create(B_nrows, B_ncols);
556     f = matrix_create(B_nrows, 1);
557     g = matrix_create(1, B_nrows);
558     h = matrix_create(1, 1);
559
560     for (i = 0; i < C_nrows; i++) {
561         for (j = 0; j < C_ncols; j++) {
562             int k;
563             for (k = 0; k < B_nrows; k++) row_indices[k] = i+k;
564             for (k = 0; k < B_ncols; k++) col_indices[k] = j+k;
565             GrB_extract(E->mat, GrB_NULL, GrB_NULL, A->mat, row_indices,
566                 B_nrows, col_indices, B_ncols, GrB_NULL);
567             E = matrix_elmul(E, B);
568             f = matrix_reduce(E, 0);
569             g = matrix_transpose(f);
570             h = matrix_reduce(g, 0);
571             matrix_setelem(C, matrix_getelem(h, 0, 0), i, j);
572         }
573     }

```

```

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 #ifndef 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/6PnxMC.
- [Gil] John Gilbert. *GraphBLAS: Graph Algorithms in the Language of Linear Algebra*. URL: <https://sites.cs.ucsb.edu/~gilbert/talks/Gilbert-27Jun2019.pdf>.