BLAStoff Language Reference Manual

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

Contents

1 Introduction

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

2 Lexical Conventions

2.1 Assignment

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

```
id = expr;
```

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

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

2.1.1 Matrix Literal Definition

A matrix literal looks as follows:

```
[row;
 row;
```

```
 ...]
```

where each `row` looks as follows:

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

where each `num` is either an integer, a decimal place number, or `inf` (or `-inf`). Here's an example:

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

which sets M as the matrix

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

. As we saw, the values inside the literal matrix can be anything $\in \mathbb{R} \cup \pm\infty$. Here's an example of using values other than integers:

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

which sets M as the matrix

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

.

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

2.1.2 Graph Definition

The graph definition looks as follows:

```
{
    (edge | int);
    (edge | int);
    ...
}
```

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

```
int -> int
```

Here's an example:

```
G = {
    0->1;
```

```
    1->0;
    1->2;
    4;
};
```

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

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

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

2.1.3 Number Definition

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

```
num
```

using the Here's an example:

```
M = 5;
```

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

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

which sets M as the matrix

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

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

2.1.4 Generator Function Definition

We also have a number of generator functions for commonly-used types of matrices so that you don't waste your time typing out a 50×50 identity matrix. This is what they look like:

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

The first is the `Zero` function, which generates a matrix with all 0s. This takes in one argument, which we will call x, a non-negative integer matrix of two possible sizes. n can be a 2×1 positive integer matrix, and the elements of the n matrix are the height and width of the zero matrix, in that order. n could also be a 1×1 matrix, in which case the zero matrix will be square, with the element in n as its height and width. Here is an example:

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

This code would result in the following matrices:

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

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

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

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

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

This would result in the following matrix:

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

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

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

This code would result in the following matrices:

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

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

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

2.1.5 String Definition

The final definiton 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.

2.1.6 Integers vs. Floats

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

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

```
a -= b;
M = M^a;
```

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

2.2 Comments

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

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

2.3 Functions

Functions in BLAStoff are defined as follows:

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

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

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

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

```
def bar1(A) {
```

```
    return;
}

def bar2(A) {
    ;
}
```

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

2.4 If statements

If/else statements, look as follows:

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

For example:

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

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

2.5 For/While Loops

For and while loops look as follows:

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

For example:

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

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

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

2.6 Operations

Operations are where BLAStoff gets more interesting.

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

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

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

This is how we implement these operators and some more:

2.6.1 Selection []

Here is the grammar for the selection operator:

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

The BLAStoff 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 integer matrices. c, d are optional and have a default value of $\begin{bmatrix}1\end{bmatrix}$. B is also optional and its default value is $\begin{bmatrix}0\end{bmatrix}$. Abstractly, the way this operator works is by taking the Cartesian product of A, B, $R = A \times B$, and for each $(j, i) \in R$, we select all the sub-matrices in M with a top-left corner at row j, column i, height of c, and width of d. (BLAStoff is 0-indexed.)

This Cartesian makes the select operator a very powerful operator that can do things like change a specific of indices, while also being general enough to allow for simple indexing. Take the following code example:

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

This would result in the following matrix:

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

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

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

This would result in:

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

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

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

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

This would result in:

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

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

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

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

This would result in:

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

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

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

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

This would result in:

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

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

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

This would result in:

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

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

2.6.2 Matrix Multiplication *

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

```
expr % expr
```

where % is the given operator.

The matrix multiplication operator * looks like the following:

```
A*B
```

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

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

This would result in:

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

2.6.3 Convolution

The convolution operator ~ looks like the following:

```
A~B
```

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

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

This would result in:

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

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

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

This would result in:

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

Or the dot product:

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

This would result in:

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

2.6.4 Element-wise Multiplication @

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

```
A@B
```

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

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

This would result in:

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

2.6.5 Element-wise Addition +

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

```
A+B
```

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

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

This would result in:

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

2.6.6 Exponentiation ^

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

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

We can say these correspond to

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

First we will look at the A^b case. In this case, A is an $n \times n$ (square) matrix and b is a 1×1 integer matrix. The output will be an $n \times n$ matrix as well. When $b \geq 0$, this operator is normal matrix exponentiation. For example:

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

This would result in:

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

When $b = -1$, this operator is the inversion of a matrix. Example:

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

This would result in:

$$B = \begin{bmatrix} -2 & 1 \\ 1.5 & 0.5 \end{bmatrix}$$

Note that unlike in the previous operators where float/integer rules follow the ones laid out in 2.1.5, here A can be an integer matrix, but A^{-1} is a float matrix. If A is not invertible, an error is thrown. Note that this is the only remotely

complex matrix algorithm that is computed directly "under the hood," with a language primitive.

When $b < -1$, then A^b is equivalent to $(A^{-1})^{|b|}$.

If we wanted to, we could allow b to be a float as well, but non-integer exponentiation is more difficult to calculate. So, we will determine later on if we want to allow this.

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

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

This would result in:

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

2.6.7 Size ||

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

```
|expr|
```

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

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

Example:

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

This would result in:

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

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

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

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

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

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

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

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

2.6.8 Vertical Concatenation :

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

```
A:B
```

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

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

This would result in:

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

2.6.8.1 A note on horizontal concatenation

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

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

2.6.9 Reduce Rows %

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

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

So, the two possible forms are

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

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

If

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

then

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

and

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

Here's a code example:

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

This would result in:

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

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

2.6.9.1 A note on matrices where $m = 0$

You may be wondering what happens if A is a matrix with 0 width! There is an answer to this incredibly important question: we would use 0 as the empty sum and 1 as the empty product. Example:

```
A = [;;]
B = +%A;
C = *%A;
```

This would result in:

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

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

2.6.9.2 A note on reduce columns

See 2.6.8.1.

2.6.10 Assignment operators *=, =, @=, +=, ^=, :=

The operator `*=`, used as follows:

```
A*=B;
```

is equivalent to

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

The same is true for the other assignment operators:

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

2.6.11 Comparisons $==, \neq >, \geq, <, \leq$

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

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

where A and B are both $m \times n$ matrices. These operations return our version of "true," $\begin{bmatrix} 1 \end{bmatrix}$ if these comparisons are hold element-wise in A and B. That, is

$\forall(j,i) \in ([0,m) \times [0,n))$, $A_{j,i} \geq B_{j,i}$, using the `>=` operator as an example. Note that $>$ and $<$ are not anti-symmetric under this definition. The one exception to the element-wise rule is `!=`, which is just logical not on `==`.

2.6.12 Semiring redefinition

You may have noticed that though we have defined a number of operations on matrices, when we are actually computing these matrix operations, in our examples the only operators we have actually used on the elements of these matrices are have been standard arithmetic $+$ and $\times$. However, we want to be able to use a number of semiring operators, such as those defined in the image above. BLAStoff allows for semiring redefinition in one of the following forms:

```
#logical
#arithmetic
#maxmin
#_
```

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

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

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

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


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

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

```
A = [1,4;
   6,3];
```

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

This would result in:

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

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

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

a = 4;
b = 3;
#logical;

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

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

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

a = 4;
b = 3;
#logical;

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

2.6.12.1 A note on matrices where $m = 0$, again

You may be wondering what happens in reduce rows if A is a matrix with 0 width now that we've redefined our semiring, as we had discussed the case with arithmetic in 2.6.9.1! Simply, each semi-ring has its own empty sum and product: $0, 1$ for `#logical` and $\infty, 0$ for `#minmax`.

2.6.13 Logical Negation !

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

```
!expr
```

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

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

This would result in:

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

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

2.7 Precedence

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

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

2.8 Keywords

BLAStoff reserves the following keywords:

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

3 More Language Details

3.1 Memory

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

```
def f(x){
    x += 1;
}
```

```
a = 1;
f(a);
a == 1; //FALSE
a == 2; //TRUE

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

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

3.2 Scope

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

```

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

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

3.3 Printing

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

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

OUTPUT:
Hello World!
```

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

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

```
print(toString(A));

OUTPUT:
1 2
3 4
```

4 Sample Code

4.1 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:

4.1.1 One

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

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

4.1.2 Horizontal Concatenation

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

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

4.1.3 Plus/Times Column Reduce

Column reduction follows similarly:

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

def timesColumnReduce(A){
    #_;
```

```
    return ((*%A)^T)^T;
}
```

4.1.4 Sum

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

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

def sum(A){
    #_;
    return plusColumnReduce(+%A);
}
```

4.1.5 Range From Vector

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

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

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

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

4.2 Graph Algorithms

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

```
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> ← graph^T ⊕.⊗ 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 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!

5 Roles

Katon Luaces: Manager
Michael Jan: System Architect
Jake Fisher: Language Guru
Jason Kao: Tester