MATVEC:

MATRIX-VECTOR COMPUTATION

FINAL REPORT

John C. Murphy
jcm2105
Programming Languages and Translators
Professor Stephen Edwards

SECTION 1: WHITE PAPER

MATVEC: MATRIX-VECTOR COMPUTATION PROGRAMMING LANGUAGE

INTRODUCTION:
Many programming languages today do not have the built-in functionality to easily handle the manipulation and computation of matrix and vector mathematics. Matrices and vectors are mathematical tools that are used in many different disciplines. Having a programming language that has easily usable tools and constructs for handling matrices and vectors can be very valuable. Although other tools exist today that can perform matrix/vector computations (such as Matlab), these tools often require an expensive license and require some time to learn the software package. This language has easy to use syntax that is set up to perform matrix and vector mathematical computations.

MAIN FEATURES:

SIMPLE:
This programming language provides easy-to-use, self-explanatory lexical syntax for dealing with matrices and vectors. The language will also provide an easy means to normalize vectors. The language will also provide flow-control constructs that will allow the programmer to loop through different code sections a fixed or conditional amount of times. Any novice programmer should be able to pick up and use the programming language rather quickly.

PORTABLE:
Since the programming language will be converted into Java byte code by ANTLR, the code will be executed by the Java Virtual Machine. Thus, this programming language can be run on any platform that is supported by Java.

HIGH-LEVEL FUNCTIONALITY:
The programmer will be able to easily input the values of matrices and vectors and use flow control structures within the language to further manipulate the program variables. The programmer will be able to add, subtract, multiply, and divide two matrices together using the easy-to-use program syntax. The programmer will also be able to normalize a vector if needed. Normalizing a vector simply means that the length of the vector (through any number of dimensions) is equal to one. The program will limit the normalizing of a vector to three-dimensional space.

DATA TYPES:
For the purposes of the computation, all values within a matrix or vector will assume that the values are real values. Any integer values that are input into the matrix will

automatically be converted into type real. Any variable that is not specifically declared will be assumed to be of type real.
A new data type will be established for matrix and vector computations. The data structure will be either of type matrix or vector. These will be declared at the beginning of the program.

ERROR CHECKING:

The translator will check that no invalid matrix computations are attempted. For example, addition and subtraction of two matrices requires that both matrices are identical in size. If the programmer attempted to perform a matrix addition on matrices of two different sizes, the compiler would return the appropriate error. The translator would also check whether the appropriate sizes for both matrices and vectors are used when multiplying or dividing.
The lexer from ANTLR will also check that the proper syntax and tokens have been used in the language before attempting to translate the program into Java byte code.

SCOPE:

For the purposes of computation time, the size of the matrices should be limited to 3 x 3 matrices. The error checker will make sure that the size of the matrices and vectors do not exceed this limit. For the purposes of demonstrating the functionality of the programming language, this should be a reasonable limitation.

CONTROL STRUCTURES:

The programming language will support flow control constructs such as the WHILE loop and FOR loop.

SAMPLE CODE:

The following code is an example of a typical program.

```
matrix m = [1 2 3 | 4 5 6 | 7 8 9];
matrix n = [4 5 6 | 9 8 7 | 3 2 1];
vector v = [5 8 4];
matrix p;
/* Basic matrix mathematics */
i = 0;
p = m + n;
normalize(v);
/* Loop control structure */
while (i < 5) {
p = p * v;
p = p + [1 1 i];
i = i + 1;
}
```

```
/* Print out the matrix */
print p;
```

POSSIBLE EXTENSIONS:

Time-permitting, the following features may be added to the language:

- Use of JFreeChart to display any three-dimensional vectors in space starting from the origin of the coordinate system.
- Input matrices and vectors from a file which would allow the program to process a number of different matrices at runtime.

CONCLUSION:

The purpose of MATVEC is to provide a simple-to-use programming language that can be used to perform mathematical operations on vectors and matrices. The language also provides a function for normalizing vectors as well. With the easy-to-use syntax of the language, a novice programmer should be able to perform various arithmetic computations on matrices and vectors.

SECTION 2: LANGUAGE REFERENCE MANUAL

Introduction

The purpose of this manual is to serve as a reliable reference manual for describing the MatVec language. This manual provides a general outline of the language and a detailed description of its grammar. The language describes the lexical conventions used by the language as well as the syntax notation, grammar, identifiers, and operators.

Lexical Conventions

Tokens

There are five classes of tokens that are used with this language. These tokens include: identifiers, keywords, constants, operators, and other separators. For the most part, white space is used to separate different tokens in the language. In the context of this language, white space refers to spaces, tabs, and new lines.

Comments

Comments are denoted by the use of the /* and */ characters (as in the C language). The /* character denotes the start of the comment and the */ character denotes the end of the comment string. The language does not support the use of nested comments. Single-line comments can be used by using the // symbol.

Identifiers

An identifier is a sequence of letters and numbers that represents a value. Like in the C language, the first character of an identifier must be a letter. The underscore is not supported as a letter in this language.

Keywords

The following identifiers are reserved as keywords in the MatVec language and may not be used for any other purpose:

const
else
endif
float
if
int
matrix
normalize
print
read
then
transpose
while

vector

Constants
A constant represents a fixed value that should be properly declared before it is used. There are four kinds of constants that are allowed in MatVec: an integer constant, a floating point constant, a vector constant, and a matrix constant.

These different types of constants correspond to the identifier types that are described in the Identifier Types section.

Meaning of Identifiers

Identifier Types
There are four basic identifier types in MatVec. The four types of identifiers that are allowed in MatVec are: integer type, floating point type, vector, and matrix. All of these types are described below:

Scalar Types:
An **integer** type is a sequence of digits that represents an integer. Only decimal (Base 10) numbers are supported in this language.

The following are examples of integers:
83, 124, -35, 0

A **real** type is a sequence of digits that contains a decimal point somewhere in the value. In other words, a real number contains an integer part, a decimal point, and a fractional part. The real number can be expressed using the integer part and/or the fractional part, but the decimal point must appear in the number. (A real number, however, cannot consist of just a decimal point.)

The following are examples of real numbers:
45.83, 7.5, 0.854, .12, 42.

Boolean Type:
A **Boolean** type is a defined variable in MatVec that can have a value of either “true” or “false”. The Boolean type is represented as “bool” in MatVec. Here is an example snippet of code that demonstrates a Boolean literal.

```
bool a, b;

a = true;
b = false;
```

Boolean values are also represented implicitly in the form of a conditional statement in MatVec. A conditional statement, (like “while” or “if” which are both described below), tests whether the condition is true or false.

Vector Type:
A vector is a mathematical expression that represents an entity with both magnitude and direction. The vector is represented by individual values along different orthogonal axes that represent the vector’s magnitude in those dimensions. (Theoretically, vectors are not restricted to 3-Dimensional space.)

A vector is expressed as a single column matrix in which the individual values are represented between brackets ‘[‘ and ‘]’ and the individual *values* are separated by the ‘|’ symbol. The following are examples of vector values:

[3 | 4 | 5]
[23 | 45.5 | 83.2]

Matrix Type:
A matrix is a mathematical expression of an array of data in which numbers are organized into rows and columns. A matrix is usually used to represent the coefficients of different equations that share common variables.

A matrix is expressed as two or more rows of data in which the individual values are represented between brackets ‘[‘ and ‘]’ the individual *rows* are separated by the ‘|’ symbol. The following are examples of matrix values:

Example 1:
[33 45 | 5 62| 43 50]

```
which represents matrix:    |32     45|
                            |5      62|
                            |43     50|
```

Example 2:
[4.5 20 82 | 94.5 53 .2 | 105 .12 9.2]

```
which represents matrix:    |4.5    20     82|
                            |94.5   52     .2|
                            |105    .12    9.2|
```

The size of matrices can be represented by their dimensions (M x N). Although the size of the matrix is arbitrary, all of the fields of the matrix must be populated with a value (even if the value is zero.)

Matrix/Vector Operations

Addition/Subtraction

Matrices and Vectors can only be added and subtracted with other matrices and vectors of equal dimensions. In other words, an M x N matrix can only be added or subtracted with another matrix that also has M x N dimensions. This same principle holds true when adding or subtracting vectors as well. An n-dimension vector can only be added or subtracted with another n-dimension vector.

Example 1:
[33 45 | 5 62 | 43 50] + [23 50.2 | 8 14 | .5 12]
which is equal to: [56 95.2 | 13 76 | 43.5 62]

Example 2:
[5 | 6 | 7.5] + [12 13 25]
which is equal to: [17 | 19 | 32.5]

Multiplication

The same rules that hold for matrix/vector multiplication also is enforced in MatVec as well. For instance, when a matrix of dimensions, A x B, is multiplied with a matrix of dimension, B x C, then a matrix of dimension A x C is produced from its product.

For instance:
[3 4 | 2 5] * [8 | 9] produces the result: [60 | 61]

As we see from the example, a 2 x 2 matrix that is multiplied with a 2 x 1 matrix (or vector in this case), produces a 2 x 1 matrix (or vector). Any two matrices that do not meet this criteria cannot be multiplied together to produce a product. Note, therefore, that two vectors cannot be multiplied together since they cannot meet this criterion.

Division

As it turns out, there is no such thing as dividing two matrices. Therefore, there is no operation in MatVec that supports this operation for matrices.

Scalar Multiple

The use of a scalar multiple operator in MatVec is also supported. The scalar multiple of a matrix is when a scalar value, s, is multiplied by a matrix A, to produce a scalar multiple of the matrix. In the result, every value of matrix A is multiplied by scalar, s.

For example, if we have matrix A = [E F | G H] and scalar s,
Then the result of (A * s) = [E*s F*s | G*s H*s].

Transposition

MatVec supports the transposition operation of a matrix or vector. Transposition is the conversion of a matrix A with dimensions, m x n, into a matrix B with dimensions, n x m. In the new matrix B, the following relationship holds with A: $A_{ij} = B_{ji}$.

For instance, transposing matrix [A B C] will produce the vector [A | B | C].

    transpose([A B C]) = [A | B | C]

For another example,

    matrix [A B | C D | E F] transposed would produce: [A C E | B D F]

    transpose([A B | C D | E F]) = [A C E | B D F]

Normalization

Normalization of a vector is the process in which the length of a vector is reduced to a magnitude of 1.

For instance, vector [3 | 4] normalized would equal vector [0.6 | 0.8]
Here is the reason. Since the length of this vector equals 5 (by Pythagorean Theorem, the length is 5), the different components of the vector are divided by this result to produce the normalized vector.

Expressions

Primary Expressions

A Primary Expression consists of identifiers, constants, or expressions that are contained within parentheses. The parentheses around the expression do not affect the outcome of the contained expression.

Assignment Expression

The assignment expression that is used in MatVec is the “=” operator. In the assignment expression, the value of the left operand of this expression, known as the lvalue, is reassigned to the result of the right operand. The lvalue of the expression, therefore, needs to be a valid identifier; it cannot be a constant.

Arithmetic Expressions

The arithmetic expression is a mathematical expression between two or more numbers or between two or more matrix/vector entities. When a mathematical operation is performed on a real number and an integer number, the integer number is promoted to a real number. Note that addition, subtraction, or division cannot be performed between a scalar number (i.e. integer or real) and a matrix/vector entity. When a scalar number and a matrix/vector entity are multiplied together, this is known as a “Scalar Multiple” of the matrix/vector and is described in the “Matrix/Vector Operations” section above.

Here are examples of arithmetic expressions in MatVec:

a = (35.5 * 2) + (45/9);
b = [3 | 4 | 5.5] + [12.3 | 8 | 23.4];

The following expression is NOT allowed:
c = [6 | 45 | 8] + 5;

Unary Plus Operator
The unary "+" operator is simply the result of the value of the operand. The operand can be either an integer or floating point number.

Unary Minus Operator
The unary "-" operator is the negative result of the value of the operand. The operand can be either an integer or floating point number.

Multiplicative Operators
The multiplicative operators of the language are *, /, and %. All of these operators require two operands and both operations are performed from left-to-right (left-associative operations). The * operator denotes multiplication and the / operator denotes division. The % operator denotes the modulus (or remainder) between two operands.

Additive Operators
The additive operators of the language are + and -. Both of these operators require two operands and the both operations are performed from left-to-right (left-associative operation). The + operator denotes multiplication and the / operator denotes division.

Relational Operators
The relational operators in the MatVec language are a subset of the relational operators that are used in the C language. Relational operators are used to determine whether a comparison between two operands is true or false and returns the result. Therefore, although the operation is left-associative, this fact by itself is not very useful. Since the expression a<b<c is parsed as (a<b)<c, this expression is invalid since (a<b) yields either true or false. The values of true or false cannot be one side of the relational operator.

These are the relational operators that are used by the MatVec language:
<, <=, >=, >

Less Than (<) operator
Less Than or Equal To (<=) operator
Greater Than (>) operator
Greater Than or Equal To (>=) operator

Equality Operators
The equality operators in the MatVec language are similar to the equality operators that are used in the C language. The equality operators are similar to the relational operators in that they will return a value of true or false.

These are the equality operators that are used by the MatVec language:
==, !=

Equal To (==) operator
Not Equal To (!=) operator

Operator Precedence and Order of Evaluation
The following table shows the precedence and order of evaluation of the different operators used in MatVec. The order of precedence shown in the table is a subset of the Operator Precedence and Associativity hierarchy from the C language.

Operator Name	**Operator Symbol**	**Operator Associativity**
Unary Operator	+ -	Right to Left
Multiplicative Operator	* / %	Left to Right
Additive Operator	+ -	Left to Right
Relational/Equality Operators	< <= >= > == !=	Left to Right
Assignment Operator	=	Right to Left

Declarations
In MatVec, identifiers and constants need to be declared before they are used. Declarations start with the identifier type followed by a list of identifiers or constants to be declared.

Here is an example of identifier declarations:
int i, j;
matrix u, v;
real g, h;

Here is an example of constant declarations:
const int x = 3, y = 5;
const matrix m = [1 2 3 | 4 5 6 | 7 8 9];

Statements
Statements are expressions in MatVec that are executed in sequential order. Statements in MatVec are similar to statements in C.

Conditional Statements

The conditional statements in the language are used to test whether an expression is true or false, and based on that condition, whether to execute designated lines of code. In this language, the conditionals in the conditional statements below can be either a relational or equality expression which is tested to be either true or false. The conditionals in the conditional statements must be included within the parentheses like in the examples shown below.

If-Else Conditional Statement
A conditional statement is an if-else statement where the code in the brackets is executed only when the “if” conditional is met. The “else” part of the conditional statement is optional and is only executed if the conditional is not met. Here is the iterative statement:

```
if (conditional) {
        statement;
        statement;
}
else {
        statement;
        statement;
}
endif
```

Iterative Conditional Statement
An iterative statement is the while loop that keeps executing a portion of code until the condition is satisfied. Here is the iterative statement:

```
while (conditional)
{
        statement;
        statement;
}
```

Functions
The following are functions that are available in MatVec:

User-Defined Functions
In MatVec, users will be able to define their own user-defined functions for the language. The users cannot use one of the reserved keywords shown at the beginning of this language reference manual for the function name. The user will be able to pass any user-defined variables to the function.

I/O Functions
```
read();
print();
```

The print() statement can be used to print out an identifier of any type in the language. The print statement can print out results such as the output of an integer, a vector, or a matrix value.

The read() statement is used to read in an identifier value from standard input and assign the input to an identifier. The read() statement takes the input which can be assigned to a variable of the proper type. The input format for the read statement should be in the proper format for the literal type to which the value is assigned.

For example, for the statement:

```
matrix A;
read(A);
```

The input format for A should read like:

```
[45 23 | 8 22]
```

Matrix/Vector Functions

```
normalize();
transpose();
```

The normalize() and transpose() functions will be implicit functions that are available for use in the MatVec Java bytecode.

The normalize() function takes a vector as its input and computes the "normalized" values of the input vector. An example of the use of the normalize() function is demonstrated in the "Matrix/Vector Operations" section. Here is another example of the normalize() function.

```
vector u, v;

u = normalize(v);
```

The transpose() function takes a matrix or vector as the input and produces the "transposed" matrix of the input matrix. An example of the use of the transpose() function is demonstrated in the "Matrix/Vector Operations" section. Here is another example of the transpose() function.

```
matrix m, n;

n = transpose(m);
```

SECTION 3: PROJECT OVERVIEW

Introduction

The planning, development, and testing of the MatVec language is described in this section of the report. The concept of the MatVec language as well as the basic structure of the language was determined during the development of the MatVec white paper. The specifics of the functionality and syntax were determined when the Language Reference Manual was written. During the development and testing stages, some of these design decisions were refined as the language was created.

Project Plan

Planning Stage

The major design specifications for constructing the syntax of this language were made during the development of the Language Reference Manual. The MatVec language has implemented all of the features that are described in the LRM. Minor modifications were made to the language during the development and testing stages of the walker. For instance, the user-defined functions need to appear before the main statement of the program. Although this has not been explicitly described in the LRM, the decision to implement user-defined functions in this manner was made during the software development stage.

Development Stages

The interpreter was designed, developed, and tested in different stages. In the first stage of the development, the parser and lexer were constructed for MatVec. The parser and lexer were both tested against various input files (i.e. the test suite) to verify that the parser and lexer properly constructed the abstract syntax tree (AST). Both the parser and lexer for MatVec were incorporated into the same grammar file (matvec.g).

In the next development stage, the basic structure of the tree walker was created and was also incorporated into the grammar file. Only the basic commands were implemented at first which did the most basic operations. More complicated functionality was added to the tree walker when the more basic functions were finally tested.

The basic matrix operations were incorporated into a MatrixCalc java class. This class contains the operations that are used to perform basic binary operations on two matrices and even unary operations (like transpose and normalize) that are incorporated into the functionality of the language.

Overall Design Structure

The tree walker uses two-dimensional arrays in Java to represent the matrix and vector objects that comprise the basic components of the language. The two other data types that are used in this language, which represent scalar mathematical values, are the integer and real number data types. These data types are passed between the branches of the AST within the context of a two-dimensional matrix. When the value of an integer value is printed, its value is first converted to an integer data type before its value is printed to standard output.

A null character, '\0', is used in the two-dimensional arrays to signify the end of a row or column for the matrix or vector. When the Java code reads the array, it stops incrementing the row or column based on whether a null character is encountered. For instance, the following is a logical representation of the array [1 2 3 | 4 5 6]:

```
| 1   2   3  \0 |
| 4   5   6  \0 |
| \0  \0  \0    |
```

The various mathematical operations that are performed on the matrix and vector entities in MatVec are actually executed by Java functions that manipulate the two-dimensional arrays that corresponding to matrix entities. This language can only represent matrices or vectors that have a maximum dimension of 100. However, the size of the matrix or vector that is used by the program does not have to be specified by the programmer.

Development and Test Environment

This project was developed on a Windows environment using Java 1.5.0_06. The ANTRL 2.7.5 package was downloaded to the PC from the www.antlr.org site. The following needs to be done when compiling the java code.

1.) Download and install Java 1.5.0_06 from the java.sun.com site.

2.) Download and install ANTLR 2.7.5 from the www.antlr.org site.
(example: C:\ANTLR)

    NOTE: An "antlr" directory should be created under this current directory when ANTLR is un-packed.

3.) Move the following four files under the C:\ANTLR directory:
Java Files: Main.java, MatrixCalc.java, SymbolTable.java
Grammar File: Matvec.g
Test Files: test1.txt, test2.txt, test3.txt, test4.txt, test5.txt

4.) Set the CLASSPATH to include the current antlr directory.
(example: CLASSPATH=C:\ANTLR)

5.) Set the PATH to include a path to the Java bin directory
(example: PATH=%PATH%:C:\j2sdk1.5.0_06\bin)

6.) Run the following commands to compile the code:

- java antlr.Tool Matvec.g
- javac Main.java MatrixCalc.java SymbolTable.java
- java Main <program file to run>

Java Coding Style:

- For this project, the following coding guidelines were followed when developing the Java code for this language:

- All major segments of the code have incorporated proper Java exception handling.

- All functional segments of the code have been commented to aid the reader in following the logic of the code

- The Java function names used with this project have been labeled with descriptive names that provide an understanding to the purpose of the function.

- Different Java programming constructs have been indented properly with the opening and closing braces for each construct aligned properly.

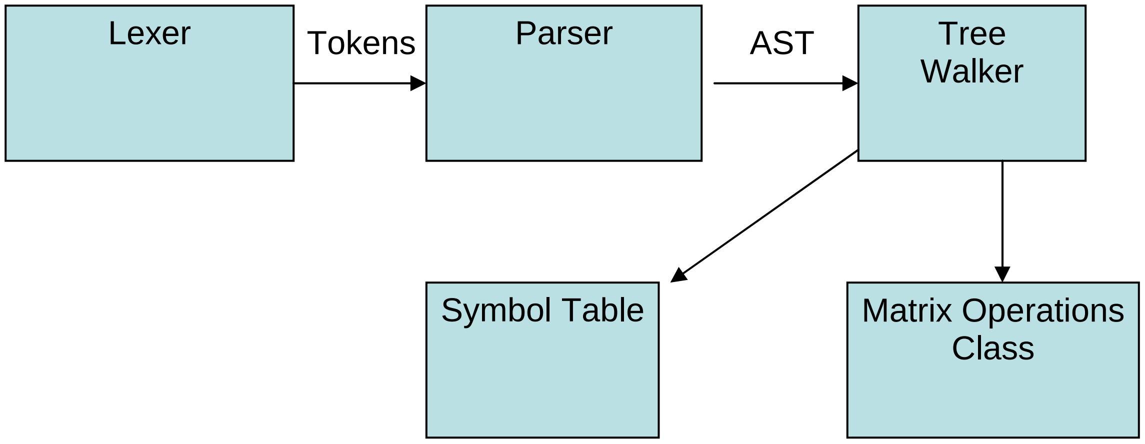
Project Timeline
The project was designed, developed, and tested within the following specified timeline:

Project Milestone	**Completed Date**
White Paper	9/27/2005
Finalize Syntax	10/18/2005
Language Reference Manual	10/20/2005
Language Reference Manual Revision	10/28/2005
Lexer Development	11/6/2005
Parser Development	11/18/2005
Tree Walker	11/25/2005
Code Development and Testing	12/10/2005
Finalize Coding and Testing	12/13/2005
Final Project Completed	12/20/2005

SECTION 4: BASIC INTERPRETER ARCHITECTURE

The interpreter is composed of the major components shown in the diagram below. These components show the components used by Antlr: the Lexer, the Parser, and the Tree Walker.

The Matrix Class Operations and the Symbol Table is Java code that is used to conduct operations on matrices/vectors and to store symbols in the symbol table.



The Symbol Table is comprised of three Hash tables in the program that are responsible for the following:

Dict: The Hash Table that is used to map the variable name with its respective value.

Dict_var: The Hash Table which is used to map the variable name with its data type. In the program, an integer has a value of 1, a real has a value of 2, a matrix has a value of 3, a vector has a value of 4, a function has a value of 5, and a Boolean has a value of 6.

Dict_const: The Hash Table which is used to map a constant name with its value. If the constant appears in this Hash Table, the interpreter will not update the value of the constant.

SECTION 5: LANGUAGE TUTORIAL

Introduction

The MatVec language is an interpreter that is used for matrix and vector computations. The syntax of the language is similar to the C language and includes a basic list of commands for the language. The following is a sample program that was used for the testing phase of this project.

Updates to Language Reference Manual:

Here are some updates that should have been included in the original Language Reference Manual that is included in this final report.

First Update:

The "invoke" command has been added to the syntax of the language. The invoke command is used to execute a function. Here is the syntax of the invoke command:

```
invoke <function name>
```

A function can also be referenced by assigning a defined variable to a defined function and having that function return a value. The second sample program will illustrate this example, but here is an example below:

```
x = hello;
```

(assuming that "x" is the variable and hello is the function name.)

Second Update:

The functions in the program need to be defined before the "main" body of the program. The functions in the program as well as the "main" section of the program need to be included in the braces as shown below. Here is an outline of a sample program:

```
function Function1 {
        <statement_1>
        …
        <statement_N>
}

function Function2 {
        <statement_1>
        …
        <statement_N>
}

…
```

```
function FunctionN {
        <statement_1>
        …
        <statement_N>
}

main {
        <statement_1>
        …
        <statement_N>
}
```

Third Update:
The Language Reference Manual (LRM) describes the fact that variables can be passed to functions. Since all variables in the program are global, these variables can be passed to the functions by simply referencing the variables in the functions. One of the sample programs in the next section will illustrate this.

Here is a sample "Hello World" program that demonstrates some of the syntax of the language. More of the syntax and functionality of the language is demonstrated in the "Test Plan" section of this report.

Sample "Hello World" Program

Here is a sample "Hello World" program:

```
function hello {
        print "Hello World";
}


main {

        definefunc hello;

/* comments are here
  this is a multiline comment */

        invoke hello;

// this is a single line comment

        print "good bye";

}
```

SECTION 6: TEST PLAN

In this project, setting up the test programs has been an on-going effort in the development of the interpreter. For every new function or statement that has been added to the Lexer, Parser, and Tree Walker, the test program was expanded to include this functionality.

For each test program, the output has been analyzed to demonstrate the functionality in the interpreter that has been implemented and tested. The final test program will demonstrate most of the functionality that has been implemented into this project.

Here are sample programs that will demonstrate these points:

Test Program 1

Here is the sample "Hello World" program from the previous section. Let's examine the output of the program below.

```
function hello {
        print "Hello World";
}


main {

        definefunc hello;

/* comments are here
   this is a multiline comment */

        invoke hello;

// this is a single line comment

        print "good bye";

}
```

This is the output that is generated from the Hello World program above:

```
( function hello { ( print Hello World ) ) ( main ( definefunc hello ) ( invoke hello ) ( print
good bye ) ) null

Hello World
good bye
```

From the example, you can see that the interpreter prints out the abstract syntax tree followed by the output of the program.

Test Program 2

Here is another sample program:

```
function test {
	real bar1;

	bar1 = 8 + 4 + x;
	print bar1;
}

function hello {
	print "sample function";
	print "more of sample function";
	return 22;
}


main {
	matrix foo;
	vector foo1, nfoo1;
	matrix foo2, foo4;
	real foo3;
	definefunc hello, test;
	bool flip;

	const integer x = 5;

/* comments are here
   this is a multiline comment */

	flip = true;
	print flip;

// this is a single line comment

	if (flip == false) {
		print "wallace";
	}
	else {
		print "grommit";
```

```
        }
        endif;

        foo3 = 5.3;
        print foo3;
        foo3 = 3 + (4 + x);

        while (foo3 <= 14) {
                print "This is a loop, foo3 is:";
                print foo3;
                foo3 = foo3 + 1;
        };

        foo = [5 3 | 4 21];
        foo1 = [3 | foo3];
        print foo;
        print foo1;
        foo4 = transpose (foo);
        print foo4;
        invoke test;
        foo3 = hello;
        print foo3;

}
```

Here is the output of the program:

```
( function test { ( real bar1 ) ( = bar1 ( + ( + 8 4 ) x ) ) ( print bar1 ) ) ( function hello { (
print sample function ) ( print more of sample function ) ( return 22 ) ) ( main ( matrix foo
) ( vector foo1 nfoo1 ) ( matrix foo2 foo4 ) ( real foo3 ) ( definefunc hello test ) ( bool flip
) ( const integer x 5 ) ( = flip true ) ( print flip ) ( if ( == flip false ) ( print wallace ) } (
print grommit ) ) ( = foo3 ( 5 .3 ) ) ( print foo3 ) ( = foo3 ( + 3 ( + 4 x ) ) ) ( while ( <=
foo3 14 ) ( print This is a loop, foo3 is: ) ( print foo3 ) ( = foo3 ( + foo3 1 ) ) ) ( = foo ( [ 5
3 | 4 21 ) ) ( = foo1 ( [ 3 | foo3 ) ) ( print foo ) ( print foo1 ) ( = foo4 ( transpose foo ) ) (
print foo4 ) ( invoke test ) ( = foo3 hello ) ( print foo3 ) ) null

True
grommit
5.3
This is a loop, foo3 is:
12.0
This is a loop, foo3 is:
13.0
This is a loop, foo3 is:
14.0
```

```
 [ 5.0 3.0 | 4.0 21.0 ]
 [ 3.0 | 15.0 ]
 [ 5.0 4.0 | 3.0 21.0 ]
17.0
sample function
more of sample function
22.0
```

First, the interpreter prints the value of the Boolean value and executes the “else” part of the “if” conditional since the value of the Boolean is “true”.

Next, we see that the interpreter executes the “while” loop until the conditional statement is evaluated as true. For each iteration of the loop, the value of foo3 is printed to demonstrate the incrementing of this variable.

Next, the various matrix and vector values are computed with one matrix assigned explicitly and the vector entity assigned using the previously assigned foo3 variable. This demonstrates that variables can be used inside matrix and vector entities that are represented in the program. We can also see that the transpose of the vector is computed and printed in the next line. Here is that section of that code:

```
        foo = [5 3 | 4 21];
        foo1 = [3 | foo3];
        print foo;
        print foo1;
        foo4 = transpose (foo);
        print foo4;
```

Finally, the program invokes the user-defined functions written in the beginning of the program. The “test” function is invoked and the foo3 variable is assigned the value that is returned from the “hello” function. Here is the output:

```
17.0
sample function
more of sample function
22.0
```

Notice that the value of bar1 in function test is computed using a value of “x” that is defined in the main body of the program.

Here is the last sample program that demonstrates some more functionality (i.e., such as the read statement) inherent in the interpreter.

Test Program 3

Here is another sample program that demonstrates the mathematics of the matrix/vector arithmetic. This program also demonstrates the functionality of the unary normalize operator in the language.

```
main {
        matrix foo;
        vector foo1, nfoo1;
        matrix foo2, foo4;
        real foo3;
        definefunc hello, test;

        const integer x = 5;

// Here is the code:

        foo3 = 5.3;
        print foo3;
        foo = [5 3 | 4 21];
        foo1 = [3 | foo3];

        print "Section 1";
        foo2 = foo * foo1;
        print foo2;

        print "Section 2";
        foo2 = foo2 * 2;
        print foo2;
        foo3 = foo3 + 4;
        print foo3;

        print "Enter in a value for foo";
        read foo;
        print foo;

        nfoo1 = normalize (foo1);
        print nfoo1;
        x = 4;

        print "good bye again";
}
```

Here is the output of the program above:

```
( function test { ( real bar1 ) ( = bar1 ( + ( + 8 4 ) x ) ) ( print bar1 ) ) ( function hello { (
print sample function ) ( print more of sample function ) ( return ( 22 .0 ) ) ) ( main (
matrix foo ) ( vector foo1 nfoo1 ) ( matrix foo2 foo4 ) ( real foo3 ) ( definefunc hello test
) ( const integer x 5 ) ( = foo3 ( 5 .3 ) ) ( print foo3 ) ( = foo ( [ 5 3 | 4 21 ) ) ( = foo1 ( [ 3 |
foo3 ) ) ( print Section 1 ) ( = foo2 ( * foo foo1 ) ) ( print foo2 ) ( print Section 2 ) ( =
foo2 ( * foo2 2 ) ) ( print foo2 ) ( = foo3 ( + foo3 4 ) ) ( print foo3 ) ( print Enter in a
value for foo ) ( read foo ) ( print foo ) ( = nfoo1 ( normalize foo1 ) ) ( print nfoo1 ) ( = x
4 ) ( print good bye again ) ) null

5.3
Section 1
 [ 30.900002 | 123.3 ]
Section 2
 [ 61.800003 | 246.6 ]
9.3
Enter in a value for foo
23 46.2 .3 | 1.2 88.1 93
 [ 23.0 46.2 0.3 | 1.2 88.1 93.0 ]
 [ 0.49259818 | 0.8702568 ]
Cannot change a constant value: x
good bye again
```

We can see in the output above that the program does a multiplication between a matrix and a vector and produces a vector as the output. The product of the matrix and vector is printed just beneath “Section 1.”

We can also see in the output above that this is an example of a scalar multiple that is described in the LRM. We can see that the vector in Section 2 is a multiple of 2 from the vector shown in Section 1 of the program. Here is the snippet of code that deals with these two vectors:

```
print "Section 1";
foo2 = foo * foo1;
print foo2;

print "Section 2";
foo2 = foo2 * 2;
print foo2;
```

We can see in this example that a value can be read from the command line and assigned to a variable in the program. The line listed above in the output of the program shows what was entered into the command line. The program then prints out the matrix, vector, or scalar value that was input into the command line. Here is the snippet of code that read in the matrix and produced the output shown:

```
print "Enter in a value for foo";
read foo;
print foo;
```

Test Program 4

Here is a sample program that incorporates almost all of the functionality that has been described in the LRM.

```
function test {
        real bar1;

        bar1 = 8 + 4 + x;
        print bar1;
}

function hello {
        print "sample function";
        print "more of sample function";
        return 22.0;
}


main {
        matrix foo;
        vector foo1, nfoo1;
        matrix foo2, foo4;
        integer foo3;
        real foo5;
        definefunc hello, test;
        bool flip;

        const integer x = 5;

/* comments are here
   this is a multiline comment */

        flip = false;
        print flip;

// this is a single line comment

        if (flip == false) {
                print "wallace";
        }
        else {
                print "grommit";
        }
```

```
endif;

foo3 = 5.3;
foo5 = 28.92;
print foo3;
print foo5;

foo3 = 3 + (4 + x);

while (foo3 <= 14) {
        print "Loop Statement";
        print foo3;
        foo3 = foo3 + 1;
};

foo = [5 3 | 4 21];
foo1 = [3 | foo3];
print foo;
print foo1;

foo2 = foo * foo1;
print foo2;

foo2 = foo2 * 2;
foo3 = foo3 + 4;
print foo3;

foo4 = transpose (foo);
print foo4;

invoke test;
foo4 = foo3 / 4;
invoke hello;

print "good bye";

foo3 = hello;
print foo3;

print "Enter in a value for foo5";
read foo5;
print foo5;

nfoo1 = normalize (foo1);
print nfoo1;
```

```
        print "good bye again";

        x = 4;
}
```

Here is the output of the program below. Note that this time the Boolean variable has been set to true so that we may see that the implicit "then" part of the "if-then-else' statement can be executed this time.

Output:

```
( function test { ( real bar1 ) ( = bar1 ( + ( + 8 4 ) x ) ) ( print bar1 ) ) ( function hello { ( print sample function ) ( print more of sample function ) ( return ( 22 .0 ) ) ) ( main ( matrix foo ) ( vector foo1 nfoo1 ) ( matrix foo2 foo4 ) ( integer foo3 ) ( real foo5 ) ( definefunc hello test ) ( bool flip ) ( const integer x 5 ) ( = flip false ) ( print flip ) ( if ( == flip false ) ( print wallace ) } ( print grommit ) ) ( = foo3 ( 5 .3 ) ) ( = foo5 ( 28 .92 ) ) ( print foo3 ) ( print foo5 ) ( = foo3 ( + 3 ( + 4 x ) ) ) ( while ( <= foo3 14 ) ( print it works ) ( print Loop Statement ) ( print foo3 ) ( = foo3 ( + foo3 1 ) ) ) ( = foo ( [ 5 3 | 4 21 ) ) ( = foo1 ( [ 3 | foo3 ) ) ( print foo ) ( print foo1 ) ( = foo2 ( * foo foo1 ) ) ( print foo2 ) ( = foo2 ( * foo2 2 ) ) ( = foo3 ( + foo3 4 ) ) ( print foo3 ) ( = foo4 ( transpose foo ) ) ( print foo4 ) ( invoke test ) ( = foo4 ( / foo3 4 ) ) ( invoke hello ) ( print good bye ) ( = foo3 hello ) ( print foo3 ) ( print Enter in a value for foo3 ) ( read foo3 ) ( print foo3 ) ( = nfoo1 ( normalize foo1 ) ) ( print nfoo1 ) ( print good bye again ) ( = x 4 ) ) null

False
wallace
5
28.92
Loop Statement
12
Loop Statement
13
Loop Statement
14
 [ 5.0 3.0 | 4.0 21.0 ]
 [ 3.0 | 15.0 ]
 [ 60.0 | 327.0 ]
19
 [ 5.0 4.0 | 3.0 21.0 ]
17.0
sample function
more of sample function
good bye
sample function
more of sample function
22
```

```
Enter in a value for foo
34.76
34.76
 [ 0.19611613 | 0.9805807 ]
good bye again
Cannot change a constant value: x
```

Note, when declaring variables, that this program lists the variables after the data type where a comma is the delimiter between the variables. This type of statement cannot be done for constant declarations. Here is the sample of the code:

```
main {
        matrix foo;
        vector foo1, nfoo1;
        matrix foo2, foo4;
        integer foo3;
        real foo5;
        definefunc hello, test;
        bool flip;

        const integer x = 5;
```

Note also that this program incorporates most of the functionality described in the LRM. In this program, the Boolean value has been changed to "true" as described above in order to demonstrate the execution of the "then" part of the "if-then-else" conditional.

Also, a value for foo, a real value, has been read into the program and assigned to foo. Note that this the value of foo is real, a scalar real value is printed, 34.76, not a matrix or vector value as shown in previous sample programs.

Finally, note that the constant value for x is used in the function "test" above. In the program, we have the following line:

```
const integer x = 5;
```

The value of x is used in the "test" function which uses this value of "x". The program correctly prints a real value of "17" based on this value of "x". However, when the program attempts to change the value of "x" to 5, an error message is printed to the screen stating that the value of "x" cannot be changed.

Test Program 5

Here is a sample program that demonstrates some remaining functionality that has been described in the LRM, namely the % operator and the integer data type.

```
main {
        matrix foo;
        vector foo1, nfoo1;
        matrix foo2, foo4;
        integer foo3;
        real foo5;
        definefunc hello, test;

        const integer x = 5;

// Here is the code:

        foo3 = 5.3;
        print foo3;
        foo = [5 3 | 4 21];
        foo1 = [3 | foo3];

        print "Section 1";
        foo2 = foo * foo1;
        print foo2;

        print "Section 2";
        foo2 = foo2 * 2;
        print foo2;
        foo5 = foo3 % 4;
        print foo5;

        print "Enter in a value for foo";
        read foo;
        print foo;

        nfoo1 = normalize (foo1);
        print nfoo1;

        print "good bye again";
}
```

Here is the output to the program:

```
( main ( matrix foo ) ( vector foo1 nfoo1 ) ( matrix foo2 foo4 ) ( integer foo3 ) (
definefunc hello test ) ( const integer x 5 ) ( = foo3 ( 5 .3 ) ) ( print foo3 ) ( = foo ( [ 5 3 |
4 21 ) ) ( = foo1 ( [ 3 | foo3 ) ) ( print Section 1 ) ( = foo2 ( * foo foo1 ) ) ( print foo2 ) (
print Section 2 ) ( = foo2 ( * foo2 2 ) ) ( print foo2 ) ( = foo3 ( % foo3 4 ) ) ( print foo3 ) (
print Enter in a value for foo ) ( read foo ) ( print foo ) ( = nfoo1 ( normalize foo1 ) ) (
print nfoo1 ) ( print good bye again ) ) null
```

```
5
Section 1
 [ 30.900002 | 123.3 ]
Section 2
 [ 61.800003 | 246.6 ]
1.3000002
Enter in a value for foo
4.5 | 8.3 | 9.7
 [ 4.5 | 8.3 | 9.7 ]
 [ 0.49259818 | 0.8702568 ]
good bye again
```

As you can see in this example, the value that is printed for foo3 this time is “5” instead of “5.3”. Since foo3 is declared as an integer data type in this program and not as a real data type, the value is truncated to “5”. Also, the mod (%) function was used to compute the remainder of foo3 divided by 4. Here is the statement from the code:

```
foo5 = foo3 % 4;
```

This statement represents 5.3 / 4 since foo3 = 5.3. This value of this statement was printed to the screen, which is the value of 1.3000002.

SECTION 7: CONCLUSIONS / LESSONS LEARNED

In this project, I have learned a number of different lessons that are worth mentioning.

- The Language Reference Manual (LRM) was developed before the nature of ANTLR and the methodology needed for designing a compiler/interpreter was completely understood. It became inherent during the development stage which syntax would be more difficult to incorporate into the interpreter.

- I understand more completely the issues and complexities that are involved with creating a programming language or translator. The use of ANTLR allowed me to work directly with the concepts involved with compiler/interpreter design without getting involved with developing a lexer and parser on my own.

- Working independently on this project has several advantages and drawbacks. As for the advantage, I do not need to coordinate with other team members with modules of the code or use a robust version control system. As a disadvantage, I do not have another team member to assist with any problems or issues that were encountered during the development stage.

- I consider this project to be a successful implementation of the syntax and functionality described in the Language Reference Manual (LRM). I have learned a great deal about compiler/interpreter design through this project and in this class. I now more fully understand the challenges involved with creating a compiler/interpreter.

SECTION 8: SOURCE CODE

Grammar File: Matvec.g

```
// John C. Murphy
// jcm 2105
// Modules: MatvecLexer, MatvecParser, MatvecWalker


class MatvecLexer extends Lexer;
options {
   testLiterals = false;
   k = 3;
   charVocabulary = '\3'..'\377';
}

PLUS      : '+' ;
MINUS     : '-' ;
MULTIPLY  : '*' ;
DIVIDE    : '/' ;
ASSIGN    : '=' ;
SEMI      : ';' ;
MOD       : '%' ;

OPENBRAC    : '[';
CLOSEBRAC   : ']';
PIPE        : '|';
DECPT       : '.';
OPENCURLY   : '{';
CLOSECURLY  : '}';
GREATERTHAN : '>';
LESSTHAN    : '<';
BANG        : '!';
COMMA       : ',';


PARENS
options {
   testLiterals = true;
}
   : '(' | ')' ;

protected LETTER : ( 'a'..'z' | 'A'..'Z' ) ;
protected DIGIT  : '0'..'9' ;
```

```
ID
options {
   testLiterals = true;
}
   : LETTER (LETTER | DIGIT | '_')* ;

// Numbers and Fractions Represented
NUMBER : (DIGIT)+;
FRACTION : DECPT (DIGIT)+;

STRING :  '"'! ( '"' '"'! | ~('"'))*  '"'!;

// White space characters
WS  :   ( ' '
      | '\t'
      | '\n' { newline(); }
      | '\r'
      ) { $setType(Token.SKIP); }
   ;


// Conditional Operations
EQUALTEST : ASSIGN ASSIGN;
NOTEQUAL : BANG ASSIGN;
GREATERTHANEQUAL : GREATERTHAN ASSIGN;
LESSTHANEQUAL : LESSTHAN ASSIGN;

// comments for multi-line comments
Comment_ml : "/*"
( options {greedy=false;}:
         (
                  ('\r' '\n') => '\r' '\n' { newline(); }
                  | '\r' { newline(); }
                  | '\n' { newline(); }
                  | ~( '\n' | '\r' )
         )
)*
"*/"
{ $setType(Token.SKIP); }
;

// comments for single-line comments
Comment_sl : "//"
(((~'\n'))* '\n' { newline(); } )
{ $setType(Token.SKIP); }
;
```

```
/////////////////////////////////////////////////////////////////////
class MatvecParser extends Parser;
options {
   buildAST = true;
   k = 3;
}

tokens {
  STATEMENTS;
}

// Overall Structure of program
file: (function)* main EOF;
function: "function"^ ID OPENCURLY (expr SEMI!)+ CLOSECURLY!;
main: "main"^ OPENCURLY! (expr SEMI!)+ CLOSECURLY!;

// Different data structure types
bool : ("true" | "false");
matrix : (NUMBER|ID) (NUMBER|ID)* (PIPE (NUMBER|ID) (NUMBER|ID)*)*;
vector : (NUMBER|ID) (PIPE! (NUMBER|ID))+;

// Conditional Statement
conditional : "("! expr (EQUALTEST^ | NOTEQUAL | GREATERTHAN^
|LESSTHAN^ |GREATERTHANEQUAL^ | LESSTHANEQUAL^) expr ")"!;


// Basic Expressions
expr
: "if"^ conditional OPENCURLY! expr (SEMI! expr)* SEMI! CLOSECURLY ("else"!
OPENCURLY! expr (SEMI! expr)* SEMI! CLOSECURLY!)? "endif"!
   | "print"^ (STRING | expr)
   | "real"^ ID (COMMA! ID)*
   | "integer"^ ID (COMMA! ID)*
   | "matrix"^ ID (COMMA! ID)*
   | "vector"^ ID (COMMA! ID)*
   | "bool"^ ID (COMMA! ID)*
   | "definefunc"^ ID (COMMA! ID)*
   | "transpose"^ "("! expr ")"!
   | "normalize"^ "("! expr ")"!
   | ID ASSIGN^ expr
   | "invoke"^ expr3
   | "return"^ expr3
   | "read"^ expr3
   | "while"^ conditional OPENCURLY! expr (SEMI! expr)* SEMI! CLOSECURLY!
```

```
    | "const"^ "integer" ID ASSIGN! expr3
    | "const"^ "real" ID ASSIGN! expr3
    | "const"^ "matrix" ID ASSIGN! expr3
    | "const"^ "vector" ID ASSIGN! expr3
    | Comment_ml
    | Comment_sl
    | expr1
    ;

expr1 : expr2 ( (PLUS^ | MINUS^) expr2 )* ;

expr2 : expr3 ( (MULTIPLY^ | DIVIDE^ | MOD^) expr3 )* ;

expr3
    : ID
    | "("! expr ")"!
    | NUMBER^ (FRACTION)?
    | FRACTION
    | MINUS^ expr3
    | OPENBRAC^ matrix CLOSEBRAC!
    | bool
    ;


///////////////////////////////////////////////////////////////////////
class MatvecWalker extends TreeParser;
options {
    k = 3;
}

// The Hash Tables used by Tree Walker
{java.util.Hashtable dict = new java.util.Hashtable();
java.util.Hashtable dict_var = new java.util.Hashtable();
java.util.Hashtable dict_const = new java.util.Hashtable();

SymbolTable symbols = new SymbolTable();
MatrixCalc mcalc = new MatrixCalc();
float[][] rvec = new float[101][101];
static int flag;
AST[] f_AST = new AST[101];
String[] f_name = new String[101];
int f_count = 0;
static int closecurly_flag;}

file: (functs)* mainfunc;
```

```
mainfunc
  : #("main"
          { float[][] vec1; }
                    (vec1=expr)+ );

functs
  : #("function" ID rvec = pred5:expr
           { float[][] mat4;
             f_AST[f_count] = pred5;
             f_name[f_count] = #ID.getText();
             f_count++;
             } );


expr returns [ float[][] vecarray = new float[101][101] ]
  { int b, c, d;
   float a, rnum, r;
   r = 0;
   float[][] vec1 = new float[101][101];
   float[][] vec2 = new float[101][101];
   float[][] vec3 = new float[101][101];
}

: #("if" vec1=pred:expr {
     try {
          AST thenpart = pred.getNextSibling();
          AST elsepart = thenpart.getNextSibling();

          a = vec1[0][0];

// If the result of the conditional is true
       if (a == 1.0) {
                    vec2 = expr(thenpart);
                    thenpart = thenpart.getNextSibling();
                    closecurly_flag = 0;
                    while (thenpart != null) {
                              if (closecurly_flag == 1)
                                        break;
                              vec2 = expr(thenpart);
                              thenpart = thenpart.getNextSibling();
                    }
          }

// If the result of the conditional is false
       else if (elsepart != null) {
                    vec2 = expr(elsepart);
```

```
                elsepart = elsepart.getNextSibling();
                while (elsepart != null) {
                        vec2 = expr(elsepart);
                        elsepart = elsepart.getNextSibling();
                }
        }

     else vec2[0][0] = 0;

    } catch(Exception e) { System.err.println("Exception: "+e); }
} )


// The different type of constant data types are described
| #("const"
        ("integer" ID vec1=expr {
                String t = #ID.getText();
                b = 1;
                dict_var.put(t, new Integer(b));
                dict_const.put(t, vec1);
                dict.put(t, vec1); }

        |"real" ID vec1=expr {
                String t = #ID.getText();
                b = 2;
                dict_var.put(t, new Integer(b));
                dict_const.put(t, vec1);
                dict.put(t, vec1);
        }

        |"matrix" ID vec1=expr {
                String t = #ID.getText();
                b = 3;
                dict_var.put(t, new Integer(b));
                dict_const.put(t, vec1);
                dict.put(t, vec1);
        }

        |"vector" ID vec1=expr {
                String t = #ID.getText();
                b = 4;

                vec2 = vec1;
                if (mcalc.CheckVector(vec2) == false)
                        System.out.println("Invalid vector value");
                else {
                        dict_var.put(t, new Integer(b));
                        dict_const.put(t, vec1);
                        dict.put(t, vec1);
                }
        } )
)


| #("while" vec1=pred3:expr {
   try {
        AST condpart = pred3;
        AST looppart = pred3.getNextSibling();
        AST startloop = looppart;
        a = vec1[0][0];

        while (a != 0) {
                looppart = startloop;
                vec2 = expr(looppart);
                looppart = looppart.getNextSibling();

// Execute while loop while children appear in AST
                while (looppart != null) {
                        vec2 = expr(looppart);
                        looppart = looppart.getNextSibling();
                }

                vec1=expr(condpart);
                a = vec1[0][0];
        }

        vec2[0][0] = 0;

   } catch(Exception e) { System.err.println("Exception: "+e); }
} )


// Different comparison tests for conditional statements.
// If conditional is true, 1 is returned
// If conditional is false, 0 is returned
| #(EQUALTEST vec1=expr vec2=expr {
        if (mcalc.CompareMatrix(vec1, vec2))
                vecarray[0][0] = 1;
        else
                vecarray[0][0] = 0;
        }
)
```

```
| #(NOTEQUAL vec1=expr vec2=expr {
        if (!mcalc.CompareMatrix(vec1, vec2))
                vecarray[0][0] = 1;
        else
                vecarray[0][0] = 0;
        }
)

| #(GREATHERTHAN vec1=expr vec2=expr {
        if ((mcalc.ScalarValue(vec1) == false)||(mcalc.ScalarValue(vec2) == false))
                vecarray[0][0] = 0;

        if (vec1[0][0] > vec2[0][0])
                vecarray[0][0] = 1;
        else
                vecarray[0][0] = 0;
        }
)

| #(LESSTHAN vec1=expr vec2=expr {
        if ((mcalc.ScalarValue(vec1) == false)||(mcalc.ScalarValue(vec2) == false))
                vecarray[0][0] = 0;

        if (vec1[0][0] < vec2[0][0])
                vecarray[0][0] = 1;
        else
                vecarray[0][0] = 0;
        }
)

| #(GREATHERTHANEQUAL vec1=expr vec2=expr {
        if ((mcalc.ScalarValue(vec1) == false)||(mcalc.ScalarValue(vec2) == false))
                vecarray[0][0] = 0;

        if (vec1[0][0] >= vec2[0][0])
                vecarray[0][0] = 1;
        else
                vecarray[0][0] = 0;
        }
)

| #(LESSTHANEQUAL vec1=expr vec2=expr {
        if ((mcalc.ScalarValue(vec1) == false)||(mcalc.ScalarValue(vec2) == false))
                vecarray[0][0] = 0;
        if (vec1[0][0] <= vec2[0][0])
                vecarray[0][0] = 1;
        else
                vecarray[0][0] = 0;
        }
)


| #("invoke" ID {
        float[][] mat4;
        int count = 0;
        AST nextnode = f_AST[count];

   try {
        while (count <= f_count) {

// Find the function that is invoked
// Each array element of f_name contains a sub-tree of AST corresponding to function
                if (f_name[count].equals(#ID.getText())) {
//                      System.out.println("Running Function: " + f_name[count]);
                        mat4 = expr(f_AST[count]);
                        nextnode = f_AST[count].getNextSibling();

                        while (nextnode != null) {
                                mat4 = expr(nextnode);
                                nextnode = nextnode.getNextSibling();
                        }

                        break;
                }
                count++;
        }

   } catch(Exception e) { System.err.println("Exception: "+e); }
} )


| #("return" vec1=expr
        {vecarray=vec1;} )


| #("print"
    ( s:STRING { System.out.println(#s.getText()); }
    | ID {

    try {
```

```
// This finds the data type of ID in the dict_var Hash Table
        if ( !(dict_var.containsKey(#ID.getText())))
                System.err.println("unrecognized: "+#ID.getText());
        b = ((Integer) dict_var.get(#ID.getText())).intValue();

// If data type is integer, ID is converted to integer before it is printed out.
        if (b == 1) {
                vec1 = ((float[][]) dict.get(#ID.getText()));
                b = (int) vec1[0][0];
                System.out.println(b);
        }

        if (b == 2) {
                vec1 = ((float[][]) dict.get(#ID.getText()));
                a = vec1[0][0];
                System.out.println(a);
        }

// If data type is vector or matrix, the PrintMatrix method is invoked.
        if ((b == 3) || (b == 4)) {
                vec1 = ((float[][]) dict.get(#ID.getText()));
                mcalc.PrintMatrix(vec1);
        }

// If data type is a boolean, print the corresponding value.
        if (b == 6) {
                vec1 = ((float[][]) dict.get(#ID.getText()));
                a = vec1[0][0];
                if (a == 1)
                        System.out.println("True");
                else
                        System.out.println("False");
        }

    } catch(Exception e) { System.err.println("Exception: "+e); }

} ) )


// This section of the code deals with the declaration of data types
// Note that more than one variable can appear after a data type
// delimited by commas.
| #("integer" ID
        {String t = #ID.getText();
        b = 1;
        dict_var.put(t, new Integer(b));
        AST nextnode = #ID.getNextSibling();
        while (nextnode != null) {
                t = nextnode.getText();
                dict_var.put(t, new Integer(b));
                nextnode = nextnode.getNextSibling();
        } } )

| #("real" ID
        {String t = #ID.getText();
        b = 2;
        dict_var.put(t, new Integer(b));
        AST nextnode = #ID.getNextSibling();
        while (nextnode != null) {
                t = nextnode.getText();
                dict_var.put(t, new Integer(b));
                nextnode = nextnode.getNextSibling();
        } } )

| #("matrix" ID
        {String t = #ID.getText();
        b = 3;
        dict_var.put(t, new Integer(b));
        AST nextnode = #ID.getNextSibling();
        while (nextnode != null) {
                t = nextnode.getText();
                dict_var.put(t, new Integer(b));
                nextnode = nextnode.getNextSibling();
        } } )

| #("vector" ID
        {String t = #ID.getText();
        b = 4;
        dict_var.put(t, new Integer(b));
        AST nextnode = #ID.getNextSibling();
        while (nextnode != null) {
                t = nextnode.getText();
                dict_var.put(t, new Integer(b));
                nextnode = nextnode.getNextSibling();
        } } )

| #("definefunc" ID
        {String t = #ID.getText();
        b = 5;
        dict_var.put(t, new Integer(b));
        dict.put(t, "function");
```

```
            AST nextnode = #ID.getNextSibling();
            while (nextnode != null) {
                        t = nextnode.getText();
                        dict_var.put(t, new Integer(b));
                        dict.put(t, "function");
                        nextnode = nextnode.getNextSibling();
            } } )

| #("bool" ID
            {String t = #ID.getText();
            b = 6;
            dict_var.put(t, new Integer(b));
            AST nextnode = #ID.getNextSibling();
            while (nextnode != null) {
                        t = nextnode.getText();
                        dict_var.put(t, new Integer(b));
                        nextnode = nextnode.getNextSibling();
            } } )


// The ASSIGN is used to declare variables to their corresponding values
| #(ASSIGN ID
            {String str = #ID.getText();}
   (vec1=expr
    {
     try {

// This checks whether the variable to be declared is a constant value.
            if (dict_const.containsKey(str))
                        System.err.println("Cannot change a constant value: "+str);

// This checks the variable type of ID
            else if ( !(dict_var.containsKey(str)))
                        System.err.println("unrecognized: "+str);

            else {
                        b = ((Integer) dict_var.get(str)).intValue();

// This checks whether the vector is valid if a vector data type is used
                        vec2 = vec1;
                        if ((b == 4) && (mcalc.CheckVector(vec2) == false)) {
                                    System.out.println("Invalid vector value");
                        }

                        else {
                                    dict.put(str, vec2);
                        }
            }

       } catch(Exception e) { System.err.println("Exception: "+e); }

} ) )


| #("read" ID {
     try {
            String str = #ID.getText();

// The GetInput() function is invoked to retrieve the user input
            vec1 = symbols.GetInput();

// The data type of ID is checked
            if ( !(dict_var.containsKey(str)))
                        System.err.println("unrecognized: "+str);
            b = ((Integer) dict_var.get(str)).intValue();

// This checks whether the vector is valid if a vector data type is used
            vec2 = vec1;
            if ((b == 4) && (mcalc.CheckVector(vec2) == false)) {
                        System.out.println("Invalid vector value");
            }

            else {
                        dict.put(str, vec2);
            }

     } catch(Exception e) { System.err.println("Exception: "+e); }
} )


// This part of the AST deals with basic matrix/vector and scalar operations
// The corresponding matrix/vector operation is invoked in each branch.
| #(PLUS vec1=expr vec2=expr {
            rvec = mcalc.AddMatrix(vec1, vec2);
//          mcalc.PrintMatrix(rvec);
            vecarray = rvec; }
)

| #(MINUS vec1=expr vec2=expr {
            rvec = mcalc.SubtractMatrix(vec1, vec2);
            vecarray = rvec; }
)
```

```
| #(MULTIPLY vec1=expr vec2=expr {
        rvec = mcalc.MultiplyMatrix(vec1, vec2);
        vecarray = rvec; }
)

| #(DIVIDE vec1=expr vec2=expr {
        rvec = mcalc.DivideMatrix(vec1, vec2);
        vecarray = rvec; }
)

| #(MOD vec1=expr vec2=expr {
        rvec = mcalc.ModMatrix(vec1, vec2);
        vecarray = rvec; }
)

// This part of the AST deals with basic matrix/vector unary operations
| #("transpose" vec1=expr {
        rvec = mcalc.TransposeMatrix(vec1);
        vecarray = rvec; }
)

| #("normalize" vec1=expr {
        rvec = mcalc.NormalizeMatrix(vec1);
        vecarray = rvec; }
)


// This part of AST reads the matrix or vector that is represented in the program
| #(OPENBRAC vec1=pred2:expr {
    try {
        c = 0;
        d = 0;
        vecarray[c][d] = vec1[0][0];
        d++;

// This part of the program keeps reading the next children after the OPENBRAC
// For each child of OPENBRAC, it invokes the expr()
        AST nextpart = pred2.getNextSibling();
        AST evennextpart = nextpart;

        while (nextpart != null) {
                flag = 0;
                vec2 = expr(nextpart);

                if (flag == 1) {
                        vecarray[c][d] = '\0';
                        d = 0;
                        c++;
                }
                else {
                        vecarray[c][d] = vec2[0][0];
                        d++;
                }

                evennextpart = nextpart.getNextSibling();
                nextpart = evennextpart;
        }
        vecarray[c][d] = '\0';

    } catch(Exception e) { System.err.println("Exception: "+e); }
} )


// Set the appropriate value whether "true" or "false" is found:
| #("true" {
        vecarray[0][0] = 1;
        vecarray[0][1] = '\0';
        vecarray[1][0] = '\0'; } )

| #("false" {
        vecarray[0][0] = 0;
        vecarray[0][1] = '\0';
        vecarray[1][0] = '\0'; } )


// The ID represents any variable or function name found in the program
| ID {
    try {
// The value of ID is checked to see whether it is defined
        if ( !(dict.containsKey(#ID.getText())))
                System.err.println("unrecognized: "+#ID.getText());

// The data type of ID is checked which is mainly used to check whether
// ID refers to a function invocation.
        else if ( !(dict_var.containsKey(#ID.getText())))
                System.err.println("unrecognized value: "+#ID.getText());

        else {
                int count = 0;
                AST nextnode = f_AST[count];
```

```
                    b = ((Integer) dict_var.get(#ID.getText())).intValue();

                    if (b == 5) {
                              while (count <= f_count) {

// The f_name array is an array of AST pointers that each point to a function
// defined in the program.
                              if (f_name[count].equals(#ID.getText())) {
//                                        System.out.println("Running Function: " + f_name[count]);
                                        vecarray = expr(f_AST[count]);
                                        nextnode = f_AST[count].getNextSibling();

// The while loop executes each statement of the function which corresponds to
// each child under the AST.
                                        while (nextnode != null) {
                                                  vecarray = expr(nextnode);
                                                  nextnode = nextnode.getNextSibling();
                                        }

                                        break;
                              }

                              count++;
                              }
                    }
                    else
                              vecarray = ((float[][]) dict.get(#ID.getText()));

          }

    } catch(Exception e) { System.err.println("Exception: "+e); }
}


| NUMBER { r = Integer.parseInt(#NUMBER.getText(), 10);

// If a number token is found, it is converted to an integer value.
// In the AST a real number has the whole number part as the branch.
// The fractional part is a child under the whole number.
// The whole number and fractional part are added together.
          AST nextnode = #NUMBER.getFirstChild();
          if (nextnode != null) {
                    String str = nextnode.getText();
                    rnum = Float.parseFloat(str);
                    a = r + rnum; }
          else
```

```
                a = r;

            vecarray[0][0] = a;
            vecarray[0][1] = '\0';
            vecarray[1][0] = '\0'; }

// The fraction (Without a whole number part) is returned in the array.
| FRACTION { a = Float.parseFloat(#FRACTION.getText());
            vecarray[0][0] = a;
            vecarray[0][1] = '\0';
            vecarray[1][0] = '\0'; }


// This flag is used to signal that the next row of the matrix or vector
// needs to be populated when a pipe "|" is encountered.
| PIPE {
          flag = 1;
          r = 0; }

| OPENCURLY { r = 0; }


// This flag signals the end of the "then" part of the "if-then-else" conditional.
| CLOSECURLY {
          closecurly_flag = 1;
          r = 0; }

;
```

Java File: Main.java

```
// John C. Murphy
// jcm 2105
// Module: Main

import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.debug.misc.ASTFrame;

public class Main {

  public static void main(String args[]) {
    try {

// Get file stream input specified as program argument
            String program_name = args[0];
            File program_obj = new File(program_name);
            FileInputStream input = new FileInputStream(program_obj);

// Instantiate the Lexer and Parser
      MatvecLexer lexer = new MatvecLexer(input);
      MatvecParser parser = new MatvecParser(lexer);
      parser.file();

// Generate the Parse Tree and print it to the screen
      AST parseTree = (AST)parser.getAST();
      System.out.println(parseTree.toStringList());
      System.out.println("");

// Invoke the Tree Walker and pass the Parse Tree as the argument
      MatvecWalker walker = new MatvecWalker();
      walker.file(parseTree);

    } catch(Exception e) { System.err.println("Exception: "+e); }
  }

}
```

Java File: MatrixCalc.java

```
// John C. Murphy
// jcm 2105
// Modules:     PrintMatrix, AddMatrix, SubtractMatrix, CompareMatrix
//              MultiplyMatrix, DivideMatrix, ModMatrix
//              ScalarMatrix, TransposeMatrix, NormalizeMatrix
//              CheckVector, ScalarValue, PrintSizeError


import java.io.*;
import java.util.*;


// Print out the matrix/vector in the form: [ a b c | d e f | g h i ]

public class MatrixCalc extends Main {
        int c,d;

        public void PrintMatrix(float[][] matrix) {

        try {
//              System.out.println("PrintMatrix");
                c = 0;
                d = 0;

                System.out.print(" [ ");
                while (matrix[c][d] != '\0') {
                        while (matrix[c][d] != '\0') {
                                System.out.print(matrix[c][d]);
                                System.out.print(" ");
                                d++;
                        }
                        d = 0;
                        c++;
                        if (matrix[c][d] != '\0')
                                System.out.print("| ");
                }
                System.out.println("]");

        } catch(Exception e) { System.err.println("Exception: "+e);}

        }


// Add two matrices/vectors/scalars together (represented as two-dimensional arrays)
// Return matrix3 which is the sum of two entities

        public float[][] AddMatrix(float[][] matrix1, float[][] matrix2) {
                float[][] matrix3 = new float[101][101];
                boolean result;

        try {
                result = CompareMatrix(matrix1, matrix2);
                if (result == false) {
                        PrintSizeError(matrix1, matrix2);
                        return matrix3;
                }

//              System.out.println("AddMatrix");
                c = 0;
                d = 0;

                while (matrix1[c][d] != '\0') {
                        while (matrix1[c][d] != '\0') {
                                matrix3[c][d] = matrix1[c][d] + matrix2[c][d];
                                d++;
                        }
                        d = 0;
                        c++;
                }

                return matrix3;

        } catch(Exception e) { System.err.println("Exception: "+e);
                                return matrix3;}
        }


// Subtract two matrices/vectors/scalars (represented as two-dimensional arrays)
// Return matrix3 which is the difference of two entities

        public float[][] SubtractMatrix(float[][] matrix1, float[][] matrix2) {
                float[][] matrix3 = new float[101][101];

        try {
//              System.out.println("SubtractMatrix");
                c = 0;
                d = 0;

                while (matrix1[c][d] != '\0') {
                        while (matrix1[c][d] != '\0') {
```

```
                        matrix3[c][d] = matrix1[c][d] - matrix2[c][d];
                        d++;
                    }
                    d = 0;
                    c++;
                }

                return matrix3;

        } catch(Exception e) { System.err.println("Exception: "+e);
                        return matrix3;}
        }


// Compare two matrices/vectors/scalars (represented as two-dimensional arrays)
// and return boolean whether the two entities are identical

        public boolean CompareMatrix(float[][] matrix1, float[][] matrix2) {

        try {
//              System.out.println("CompareMatrix");
                c = 0;
                d = 0;
                while (matrix1[c][d] != '\0') {
                    while (matrix1[c][d] != '\0') {
                        if (matrix2[c][d] == '\0')
                            return false;
                        d++;
                    }
                    if (matrix2[c][d] != '\0')
                        return false;
                    d = 0;
                    c++;
                }
                if (matrix2[c][d] != '\0')
                    return false;

                return true;

        } catch(Exception e) { System.err.println("Exception: "+e);
                        return true;}
        }


// Multiply two matrices/vectors/scalars (represented as two-dimensional arrays)
// Return matrix3 which is the product of two entities
```

```
	public float[][] MultiplyMatrix(float[][] matrix1, float[][] matrix2) {
		float[][] matrix3 = new float[101][101];
		float[][] matrix4 = new float[101][101];
		int x;
		float sum;

	try {
//		System.out.println("MultiplyMatrix");

		if ((ScalarValue(matrix1) == true)&&(ScalarValue(matrix2) == false)) {
			matrix3 = ScalarMatrix(matrix1, matrix2);
			return matrix3;
		}

		if ((ScalarValue(matrix1) == false)&&(ScalarValue(matrix2) == true)) {
			matrix3 = ScalarMatrix(matrix2, matrix1);
			return matrix3;
		}

		if ((ScalarValue(matrix1) == true)&&(ScalarValue(matrix2) == true)) {
			matrix3[0][0] = matrix1[0][0] * matrix2[0][0];
			matrix3[0][1] = '\0';
			matrix3[1][0] = '\0';
			return matrix3;
		}

		c = 0;
		d = 0;
		while (matrix1[c][0] != '\0') {
			while (matrix2[0][d] != '\0') {

				x = 0;
				sum = 0;
				while (matrix1[c][x] != '\0') {
					if (matrix2[x][d] == '\0') {
						PrintSizeError(matrix1, matrix2);
						return matrix4;
					}
					sum = matrix1[c][x]*matrix2[x][d] + sum;
					x++;
				}
				matrix3[c][d] = sum;
				d++;
			}
```

```
                    d = 0;
                    c++;
                }

                return matrix3;

            } catch(Exception e) { System.err.println("Exception: "+e);
                                    return matrix3;}
            }


// Divide two scalars (represented as two-dimensional arrays)
// Return matrix3 which is the quotient of two scalars

            public float[][] DivideMatrix(float[][] matrix1, float[][] matrix2) {
                float[][] matrix3 = new float[101][101];

            try {
//              System.out.println("DivideMatrix");

                if ((ScalarValue(matrix1) == false)&&(ScalarValue(matrix2) == false)) {
                    System.out.println("These matrices cannot be used with divide:");
                    PrintMatrix(matrix1);
                    PrintMatrix(matrix2);
                    return matrix3;
                }

                if ((ScalarValue(matrix1) == false)&&(ScalarValue(matrix2) == true)) {
                    System.out.println("This matrix cannot be used with divide:");
                    PrintMatrix(matrix1);
                    return matrix3;
                }

                if ((ScalarValue(matrix1) == true)&&(ScalarValue(matrix2) == false)) {
                    System.out.println("This matrix cannot be used with divide:");
                    PrintMatrix(matrix2);
                    return matrix3;
                }

                if (matrix2[0][0] == 0) {
                    System.out.println("Divide by zero is not allowed");
                    return matrix3;
                }

                if ((ScalarValue(matrix1) == true)&&(ScalarValue(matrix2) == true)) {
                    matrix3[0][0] = matrix1[0][0] / matrix2[0][0];
```

```
                        matrix3[0][1] = '\0';
                        matrix3[1][0] = '\0';
                        return matrix3;
                }

                return matrix3;

        } catch(Exception e) { System.err.println("Exception: "+e);
                                return matrix3;}
        }


// Divide two scalars (represented as two-dimensional arrays)
// Return matrix3 which is the mod (remainder) of two scalars

        public float[][] ModMatrix(float[][] matrix1, float[][] matrix2) {
                float[][] matrix3 = new float[101][101];

        try {
//              System.out.println("ModMatrix");

                if ((ScalarValue(matrix1) == false)&&(ScalarValue(matrix2) == false)) {
                        System.out.println("These matrices cannot be used with divide:");
                        PrintMatrix(matrix1);
                        PrintMatrix(matrix2);
                        return matrix3;
                }

                if ((ScalarValue(matrix1) == false)&&(ScalarValue(matrix2) == true)) {
                        System.out.println("This matrix cannot be used with divide:");
                        PrintMatrix(matrix1);
                        return matrix3;
                }

                if ((ScalarValue(matrix1) == true)&&(ScalarValue(matrix2) == false)) {
                        System.out.println("This matrix cannot be used with divide:");
                        PrintMatrix(matrix2);
                        return matrix3;
                }

                if (matrix2[0][0] == 0) {
                        System.out.println("Divide by zero is not allowed");
                        return matrix3;
                }

                if ((ScalarValue(matrix1) == true)&&(ScalarValue(matrix2) == true)) {
```

```
                matrix3[0][0] = matrix1[0][0] % matrix2[0][0];
                matrix3[0][1] = '\0';
                matrix3[1][0] = '\0';
                return matrix3;
            }

            return matrix3;

        } catch(Exception e) { System.err.println("Exception: "+e);
                    return matrix3;}
        }


// This function is invoked by the MultiplyMethod() function
// which is used to perform a scalar multiple (scalar value times matrix/vector)

        public float[][] ScalarMatrix(float[][] scalar1, float[][] matrix1) {
            float[][] matrix2 = new float[101][101];
            float r;

        try {
            r = scalar1[0][0];
            c = 0;
            d = 0;
            while (matrix1[c][d] != '\0') {
                while (matrix1[c][d] != '\0') {
                    matrix2[c][d] = r * matrix1[c][d];
                    d++;
                }

                d = 0;
                c++;
            }

            return matrix2;

        } catch(Exception e) { System.err.println("Exception: "+e);
                    return matrix2;}
        }


// This function represents a unary operator which takes the transposition
// of a matrix or vector.

        public float[][] TransposeMatrix(float[][] matrix1) {
            float[][] matrix2 = new float[101][101];
        try{
//              System.out.println("TransposeMatrix");
                c = 0;
                d = 0;
                while (matrix1[c][d] != '\0') {
                        while (matrix1[c][d] != '\0') {
                                matrix2[d][c] = matrix1[c][d];
                                d++;
                        }
                        matrix2[d][c] = '\0';

                        d = 0;
                        c++;
                }
                matrix2[d][c] = '\0';

                return matrix2;
        } catch(Exception e) { System.err.println("Exception: "+e);
                                return matrix2;}
        }


// This function represents a unary operator which takes the normalization
// of a vector.  The function uses CheckVector() to make sure that the
// array represents a vector and not a matrix.

        public float[][] NormalizeMatrix(float[][] vector1) {
                float[][] vector2 = new float[101][101];
                float sum, result;

        try {
//              System.out.println("NormalizeMatrix");

                if (CheckVector(vector1) == false) {
                        System.out.println("Invalid vector");
                        return vector2;
                }

                c = 0;
                sum = 0;
                while (vector1[c][0] != '\0') {
                        sum = (vector1[c][0] * vector1[c][0]) + sum;
                        c++;
                }
```

```
		result = (float) Math.sqrt(sum);

		c = 0;
		while (vector1[c][0] != '\0') {
			vector2[c][0] = (vector1[c][0])/ result;
			c++;
		}

		vector2[c][0] = '\0';

		return vector2;

	} catch(Exception e) { System.err.println("Exception: "+e);
				return vector2;}
	}


// This function checks whether the array represents a vector.  The function
// returns "true" if the array represents a vector and "false" otherwise.

	public boolean CheckVector(float[][] matrix) {

	try {
//		System.out.println("CheckVector");

		if (ScalarValue(matrix) == true)
			return false;

		c = 0;
		while (matrix[c][0] != '\0') {
			if (matrix[c][1] != '\0') {
				return false;
			}
		c++;
		}

		return true;

	} catch(Exception e) { System.err.println("Exception: "+e);
				return true;}
	}


// This function checks whether the array represents a scalar value and not
// a vector or matrix.  This means that the array only has a single non-null
// value at matrix[0][0] and a null value at matrix[0][1] and matrix[1][0].
```

```
        public boolean ScalarValue(float[][] matrix) {

        try {
                if ((matrix[0][1] == '\0')&&(matrix[1][0] == '\0'))
                        return true;

                return false;

        } catch(Exception e) { System.err.println("Exception: "+e);
                                return false;}
        }


// This function is invoked by other functions to print the fact that the
// given matrices do not match in size and cannot be added or subtracted.

        public void PrintSizeError(float[][] matrix1, float[][] matrix2) {

        try {
                System.out.println("The following matrices do not match in size");
                PrintMatrix(matrix1);
                PrintMatrix(matrix2);

        } catch(Exception e) { System.err.println("Exception: "+e);}

        }

}
```

Java File: SymbolTable.java

```
// John C. Murphy
// jcm 2105
// Modules: SymbolTable, GetInput

import java.io.*;
import java.util.*;

public class SymbolTable extends Main {
        Hashtable hashvar;
        Hashtable hashtype;
        Hashtable hashconst;

// These Hash Tables have been incorporated into the grammar file
        public SymbolTable() {
                hashvar = new Hashtable();
                hashtype = new Hashtable();
                hashconst = new Hashtable();
        }

// This function retrieves the input from the user and parses the input
        public float[][] GetInput() {
                float[][] vec = new float[101][101];
                float r;
                int i, lowerbound, upperbound;
                int row, column;
                char c;
                boolean value;

                try{

// This part retrieves the input from the input stream buffer
                        BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
                        String userinput = in.readLine();

                        vec[0][0] = '\0';
                        vec[0][1] = '\0';
                        vec[1][0] = '\0';

// The variables are initialized before parsing the input buffer.
// The boolean "value" is used to determine whether the parser is in the process
// of reading in a number and whether the parser just finished parsing a number.
// If the parser just finished parsing a number, the substring is converted to a float
// and input to the array.
```

```
                        lowerbound = 0;
                        upperbound = 0;
                        row = 0;
                        column = 0;
                        value = false;

// This for-loop increments through the input and parses each character of the buffer.
                        for (i=0; i<userinput.length(); i++) {

                                if (userinput.charAt(i) == '[') {
                                        lowerbound++;
                                        upperbound++;
                                }

                                else if (userinput.charAt(i) == ']')
                                        break;

                                else if (userinput.charAt(i) == '|') {
                                        row++;
                                        column = 0;
                                }

                                else if (userinput.charAt(i) == ' ') {

                                        if (value == true) {
                                                upperbound = i;
                                                r =
Float.parseFloat(userinput.substring(lowerbound,upperbound));
                                                vec[row][column] = r;
                                                vec[row][column+1] = '\0';
                                                vec[row+1][column] = '\0';
                                                value = false;
                                                column++;
                                        }
                                        else
                                                lowerbound = i;
                                }
                                else {
//                                      System.out.println(userinput.charAt(i));
                                        if (value == true)
                                                upperbound = i;
                                        if (value == false) {
                                                upperbound = i;
                                                lowerbound = i;
                                                value = true;
```

```
                    }
                }
            }

// This code below is needed to process the last number read from the buffer.
            if (value == true) {
                upperbound++;
                r =
Float.parseFloat(userinput.substring(lowerbound,upperbound));
                vec[row][column] = r;
                vec[row][column+1] = '\0';
                vec[row+1][column] = '\0';
                value = false;
            }

            return vec;

        } catch(Exception e) { System.err.println("Exception: "+e);
                    return vec; }
    }

}
```