Matrix and Scientific Computing Language
(MSCL)

Final Report

COMS W4115 – Programming Languages and Translator

Rui Chen

12/21/2010

I. Introduction

The purpose of the Matrix and Scientific Computing Language (MSCL) is to provide a tool for performing basic matrix operations and scientific calculations. The inspiration for MSCL comes from MATLAB. For this project, a small subset of MATLAB's direct matrix manipulation capabilities has been implemented in MSCL. In addition, various build-in functions and control structures have been implemented to allow MSCL to solve simple numerical problems such as ordinary differential equations (ODEs).

The first goal in the development of MSCL is to demonstrate that it is capable of manipulating matrices directly using operators such as "+" or "*". The second goal is to demonstrate that MSCL can be used to solve first and second order ordinary differential equations (ODE) for initial value problems (IVP).

The syntax of MSCL is similar to that of MATLAB. To construct a matrix, the user first declares a variable of type *matrix*:

```
matrix M;
```

Then specify the size of the matrix:

```
M=newmat[2,2];
```

Then specify the value of the matrix;

```
M=[0,2;3,1];
```

This constructs the matrix:

$$M = \begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix}$$

Operation on this matrix can be done directly using various math operators. For example:

M+*M* results in $\begin{bmatrix} 2 & 4 \\ 6 & 2 \end{bmatrix}$

*M***3* results in $\begin{bmatrix} 3 & 6 \\ 9 & 3 \end{bmatrix}$

M./*M* results in $\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$

MSCL is also capable of numerically solving initial value problems (IVP) that that can be written in the following form:

$$y' = f(t, y) \qquad \text{(eqn. 1)}$$

where y' denotes the first derivative of y. MSCL allows the user to define the algorithm and data required for solving a particular ODE. It also has loop and logical control

structures, screen display and file I/O capabilities. Using MSCL, a user can simulate a system such as a simplified suspension system of a car and investigate the effect of different parameters such as the damping coefficient and natural frequency on the behavior of the system.

More details on how to use MSCL to solve initial value problems and perform matrix operations are included in the section which gives a tutorial on how to program in MSCL. In addition, the language tutorial also shows the user how to compile the MSCL compiler as well as MSCL source codes.

II. Language Tutorial

1. Compiling MSCL Compiler and MSCL Programs

The MSCL compiler is programmed using OCaml. A program written in MSCL is first translated to Java source code. Then the Java source code can be run on Java Virtual Machine (JVM). The following source files are included:

scanner.mll (scanner)
parser.mly (parser)
ast.mli (abstract syntax tree)
semantics.ml (semantics analyzer)
javawriter.ml (java translator)
mscl.ml (main program)

The following shell script is used to compile the compiler source code in Windows XP:

```
ocamllex scanner.mll
ocamlyacc parser.mly
ocamlc -c ast.mli
ocamlc -c parser.mli
ocamlc -c scanner.ml
ocamlc -c parser.ml
ocamlc -c semantics.ml
ocamlc -c javawriter.ml
ocamlc -c mscl.ml
ocamlc -o mscl.exe parser.cmo scanner.cmo semantics.cmo javawriter.cmo mscl.cmo
```

This script produces the MSCL compiler executable mscl.exe. This script is included with the MSCL compiler source code as a batch file – make.bat. The MSCL source also includes jama.jar, which contains the open-source Java Matrix Package (JAMA)

developed by MathWorks and NIST. The Java code generated by the MSCL compiler uses jama.jar for matrix operations.

The following syntax is used to compile a program written in MSCL:

mscl [-analyze] input_file

If the optional "–analyze" parameter is entered, MSCL will only run the semantics analyzer without translating the input file to Java.

To run a MSCL program named test.mscl, the following commands are needed:

Compile MSCL source code:

c:\test>mscl test.mscl

test.java has been outputted

Compile translated Java source code:

c:\test>javac -cp <path of jama.jar> test.java

Run the program:

c:\test>java -cp <path of jama.jar>;. test

The class path of jama.jar must be included to compile the translated java code.

2. Writing a MSCL Program

2.1. MSCL Program Structure

The following codes illustrate the structure of a MSCL program:

```
 1:  int a; %global variable declaration
 2:  function matrix o=foo(matrix b) %function declaration
 2:      o=b*2;
 4: end
 5: float d;
 6: function main()
 7:      matrix c; %local variable declaration
 8:      c=[1,2;3,4];
 9:      c=foo(c);
10:      disp(c);
11: end
```

Program 2.1

The body of this MSCL program consists of global variable declarations *a* and *d* and function declarations foo and main. A MSCL program must have a main() function from which program execution starts. Global variable and function declarations can be done in any order. Within each function, all local variables are declared in the beginning. "%" marks the beginning of a comment. Anything

between "%" and a new line character is ignored by the compiler. ";" is used to separate each statement. The semicolon is not required at the end of an *end* statement.

2.2. Variable Declaration and Assignment

As shown in Program 2.1, global variable can be declared in any order among function declarations. Local variables must be declared either at the beginning of a function or in a function's parameter list before they are assigned a value and used in the program. For example, the following program shows how to declare and assign variables a, b, c, d, and e.

```
 1: function main()
 2:     int a;
 3:     float b;
 4:     boolean c;
 5:     matrix d;
 6:     string e;
 7:     a=0;
 8:     b=1.0;
 9:     c=true;
10:     d=[1,2;3,4];
11:     e="abc";
12: end
```

Program 2.2

Variable assignment cannot be done at the same time the variable is declared. The following assignment is not correct: int a=1.0;. Variable declarations cannot be mixed with other statements. For example, moving line 3 between line 7 and 8 is incorrect. In addition, variable or literal of one type cannot be assigned to a variable of different type;.

Program 2.2 shows the declaration and assignment of the five types of variables used in MSCL: *int, float, matrix, boolean,* and *string*. The definition of the data types can be found in the language reference manual in the next section. In the matrix assignment of line#10, comma separates elements in the same row, and semicolon separates different rows. Thus, the result of line#10 is the assignment of the following matrix to *d*:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Another way to initialize and assign a matrix is the following:

```
d=newmat(3,2);
```

This set variable d to a matrix of zeros with 3 rows and 2 columns.
After a variable is declared, it is assigned a default value. The following lists the default value of all five data types:

```
int : 0
float : 0.0
matrix : [0]
boolean : false
string : “”
```

2.3. Math Operations

MSCL support various math operations between integers, floats and matrices. The following program gives some examples of supported math operations:

```
1:   function main()
2:       float r;
3:       float pi;
4:       matrix M;
5:       matrix b;
6:       M=[1,2;3,4];
7:       b=[1;2];
8:       pi=3.1415926536;
9:       r=abs(-1.0)*log(exp(2.0))*log10(10.)-(sin(pi)+cos(pi)+tan(pi));
10:      r=1+1.2*1.2-2/2.*4.0+3^2^2+4^(1/2)+sqrt(9)*(1+2-3/6);
11:      M=M*b;
12:      b=b*pi;
13:      disp("r=");
14:      disp(r);
15:  end
```

Program 2.3

The meaning of the operators are the same as those used in mathematics and other programming languages. As shown in line#10, for example, * is multiplication, + is addition, - is subtraction, / is division, and ^ is power. Depending on the operator used, the two operands do not have to have the same type. For example, for the multiplication operator * can be used to multiply, float with int, float with matrix, and int with matrix. If operands consist of both

integer and float data types as in line#10, the result will have type float. All matrix operations result in matrices containing values of type float. The detail of each operator is described in the next section. As shown in line#9, MSCL also supports various math functions such as trigonometric and exponential functions. The details of all build-in functions can be found in the next section.

Various basic matrix operations such as matrix multiplication, division, inverse, transpose, element-by-element multiplication, element-by-element division are also supported.

2.4. Control Flow Statements

2.4.1. If … else … end

```
 1:  function main()
 2:     int x;
 3:     int y;
 4:     x=1;
 5:     y=2;
 6:      if x==1
 7:              x=x+1;
 8:      else
 9:              x=x-1;
10:     end
11:    disp(x);
12:  end
```

Program 2.4.1

In Program 2.4.1, in line#5 thru #10, if x is equal to 1 then, x is set to x+1. Otherwise, x is set to x-1. The *else* statement is not optional. However, the statement between *else* and *end* is optional.

2.4.2. if… elseif…else … end

```
 1:     function main()
 2:              int x;
 3:              int y;
 4:              x=3;
 5:              y=0;
 6:               if x==1
 7:                       y=1;
```

```
 8:          elseif x==2
 9:                  y=2;
10          else
11:                 y=3;
12:         end
```

Program 2.4.2

In line#6 thru #12, if x is equal to 1, then y is set to 1. If x is equal to 2, then y is set to 2. Otherwise, y is set to 3. The *else* statement is not optional however, the statements between else and end is optional.

2.4.3. for Statement

```
1:    function  main()
2:          int x;
3:          int i;
4:          i=0;
5:          for i=10:-1:0
6:             x=x+1;
7:          end
8:          disp(x);
9:    end
```

Program 2.4.3

In line#5 thru #7, the *for* statement iterates the value of *i,* from 10 to 0 inclusive, incrementing *i* by -1 each time step. In line#5, the first number after i is the starting value, second number is the step size, and the last number is the final value. The iteration variable i can be either type *int* or *float*.

2.4.4. while Statement

```
1:    function main()
2:          int i;
3:          i = 5;
4:          while i > 0
5:                  i = i – 1;
6:          end
7:          disp("i="+i);
8:    end
```

Program 2.4.4

The while statement in line#4 thru #5 execute the statement *i = i – 1; until i* is no longer greater than 0. The statements between *while* and *end* will execute as long as the logical expression next to *while* is true.

2.5. Functions

```
1:  function int m = test (int x, int y, int z)
2:          m=x+y+z;
3:  end
4:  function  main ()
5:          int x;
6:          int y;
7:          int z;
8:          x=1;
9:          y=2;
10:         z=3;
11:         disp(test(x,y,z));
12: end
```

Program 2.4.5

Line#1 to #3 defines a function *test* that takes *int* x, *y* and z and returns an *int* m. The return variable is declared after *function,* and no explicit return statement is needed. A function can only return one variable. As shown in line#11, in the function call, the type of the parameters passed to the function must match those specified in the function declaration.

2.5.1. I/O Functions

```
1:  function main()
2:          matrix m;
3:          matrix s;
4:          int i;
5:          m=newmat(2,6);
6:          for i=0:5
7:                  m[0,i]=i;
8:          end
9:          for i=0:5
10:                 m[1,i]=i*2;
11:         end
```

```
12:         save(m,"msaveTest");
13:         s=load("msaveTest");
14:         disp(s);
15:     end
```

Program 2.6.1

Program 2.6.1 initializes a matrix m, saves matrix m in a file "msaveTest.mat" (line#12), load the file back into memory (line#13) and display the matrix on the screen(line#14). To control the format of the screen output, the following statement can be used:

setFormat("short") %4 deicmal places
setFormat("shortE") %4 decimal places in scientific notation
setFormat("long") %10 decimal places
setFormat('longE") %10 decimal places in scientific notation

2.6. Sample Programs

2.6.1. Vector Rotation

```
float pi;
function main()
%test matrix operations by testing a rotate vector function
    matrix v;
    pi=3.1415926;
    v=[0,0,1]; %unit vector pointing in z direction
    disp(rotatex(v,30.0));
    %rotate about x axis 30 deg, display resulting rotated vector
end

function matrix v = rotatex(matrix m, float angle)
    matrix Rx;
    Rx=[1,0,0;0,cos(angle*pi/180.0),-
    sin(angle*pi/180.0);0,sin(angle*pi/180.),cos(angle*pi/180.0)];
    v=m*Rx;
end
```

Output:

```
  0.0000    0.5000    0.8660
```

Program 2.6.1

Program 2.6.1 first initializes a vector *v* to a unit vector point in the +z direction ([x=0, y=0, z=1]). v and 30.0 is passed to rotatex function which takes a matrix, and an angle as arguments and return a vector which results from rotating v by the specified angle. The rotated vector is then displayed on the screen.

2.6.2. Solving 1st order ODE using 4th order Runge Kutta Method

The following program solves the following first order ODE from t=0 to t=2 using 50 steps:

$$\frac{dy}{dt} = t^2 - y, \quad y(0) = 1 \qquad \text{(eqn. 2)}$$

Then the numerical solution, the exact solution and the percentage error are stored in a matrix. The matrix are displayed on the screen and saved in a file named MoutputRK4.mat.

```
function float output=dy(float t, float y)
    %differential equation
    output=t^2-y;
end

function float output=y(float t)
    %Exact solution of dy
    output=-exp(-t)+t^2-2*t+2;
end

function matrix E=RK4(float a, float b, float ya, int N)
%solves the diff eq using 4th order Runge Kutta method
%a is the initial value of the independent variable
%b is the end value of the independent variable
%ya is the initial value of the  dependent variable
%N is the number of steps
%output is a matrix with independent variable in the first column
%and dependent variable in the second column
    float h;
    float k1;
    float k2;
    float k3;
    float k4;
    int i;
```

```
        E=newmat(N+1,2);
        h=(b-a)/N;
        E[0,0]=a;

        for i=1:N
                E[i,0]=E[i-1,0]+h;
        end

        E[0,1]=ya;
        for i=0:N-1
                k1=h*dy(E[i,0],E[i,1]);
                k2=h*dy(E[i,0]+h/2., E[i,1]+k1/2.);
                k3=h*dy(E[i,0]+h/2., E[i,1]+k2/2.);
                k4=h*dy(E[i,0]+h,E[i,1]+k3);
                E[i+1,1]=E[i,1]+(k1+2*k2+2*k3+k4)/6.0;
        end

end

function main()
%this function solves the differential equation dy using 4th order
%Runge Kutta method and displays the RK4 solution with the exact
%solution and percentage error
        matrix R; %Solution using RK4 method
        matrix Moutput; %output matrix
        float Rexact; %exact solution
        int n;
        int i;

        R=RK4(0., 2., 1., 50);

        n=height(R);
        disp(n);
        Moutput=newmat(n,4);
        for i=0:n-1
                Rexact=y(R[i,0]);
                Moutput[i,0]=R[i,0]; %t
```

```
        Moutput[i,1]=R[i,1]; %Euler method solution
        Moutput[i,2]=Rexact; %exact solution
        Moutput[i,3]=(Moutput[i,2]-
        Moutput[i,1])/Moutput[i,2]*100.0; % percent error
    end
    disp("    t      Euler     Exact     %Error");
    disp(Moutput);
    save(Moutput,"MoutputRK4");
end
```

Program 2.6.2

output:

t	RK4	Exact	%Error
0.0000	1.0000	1.0000	0.0000
0.0400	0.9608	0.9608	-0.0000
0.0800	0.9233	0.9233	-0.0000
0.1200	0.8875	0.8875	-0.0000
0.1600	0.8535	0.8535	-0.0000
0.2000	0.8213	0.8213	-0.0000
0.2400	0.7910	0.7910	-0.0000
0.2800	0.7626	0.7626	-0.0000
0.3200	0.7363	0.7363	-0.0000
0.3600	0.7119	0.7119	-0.0000
0.4000	0.6897	0.6897	-0.0000
0.4400	0.6696	0.6696	-0.0000
0.4800	0.6516	0.6516	-0.0000
0.5200	0.6359	0.6359	-0.0000
0.5600	0.6224	0.6224	-0.0000
0.6000	0.6112	0.6112	-0.0000
0.6400	0.6023	0.6023	-0.0000
0.6800	0.5958	0.5958	-0.0000
0.7200	0.5916	0.5916	-0.0000
0.7600	0.5899	0.5899	-0.0000
0.8000	0.5907	0.5907	-0.0000
0.8400	0.5939	0.5939	-0.0000
0.8800	0.5996	0.5996	-0.0000
0.9200	0.6079	0.6079	-0.0000
0.9600	0.6187	0.6187	-0.0000

1.0000	0.6321	0.6321	-0.0000
1.0400	0.6481	0.6481	-0.0000
1.0800	0.6668	0.6668	-0.0000
1.1200	0.6881	0.6881	-0.0000
1.1600	0.7121	0.7121	-0.0000
1.2000	0.7388	0.7388	-0.0000
1.2400	0.7682	0.7682	-0.0000
1.2800	0.8004	0.8004	-0.0000
1.3200	0.8353	0.8353	-0.0000
1.3600	0.8729	0.8729	-0.0000
1.4000	0.9134	0.9134	-0.0000
1.4400	0.9567	0.9567	-0.0000
1.4800	1.0028	1.0028	-0.0000
1.5200	1.0517	1.0517	-0.0000
1.5600	1.1035	1.1035	-0.0000
1.6000	1.1581	1.1581	-0.0000
1.6400	1.2156	1.2156	-0.0000
1.6800	1.2760	1.2760	-0.0000
1.7200	1.3393	1.3393	-0.0000
1.7600	1.4056	1.4056	-0.0000
1.8000	1.4747	1.4747	-0.0000
1.8400	1.5468	1.5468	-0.0000
1.8800	1.6218	1.6218	-0.0000
1.9200	1.6998	1.6998	-0.0000
1.9600	1.7807	1.7807	-0.0000
2.0000	1.8647	1.8647	-0.0000

The text output and Figure 1 show that the MSCL program worked as intended and the RK4 solution almost exactly match the exact analytical solution.

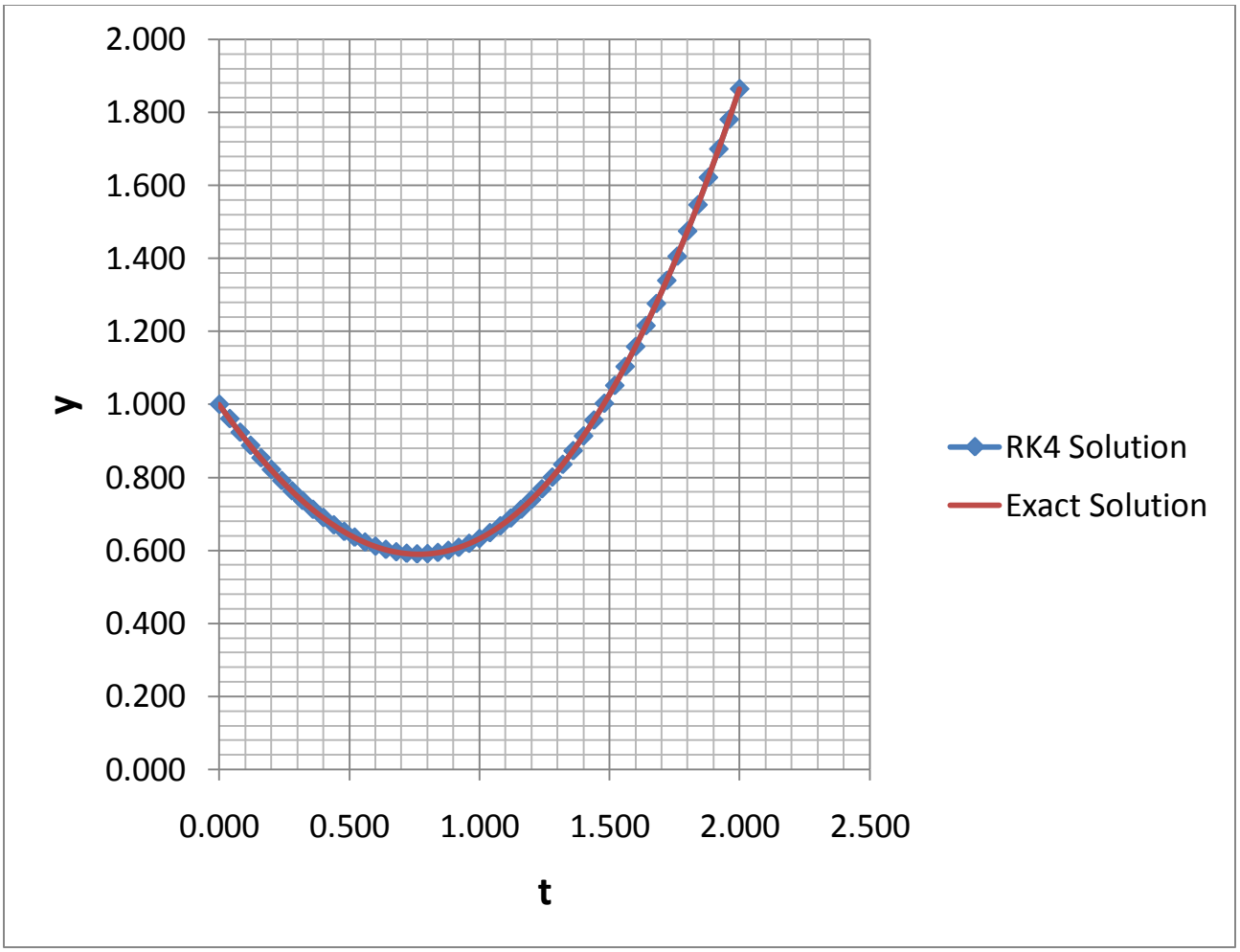


Figure 1: Comparison of RK4 Solution to Exact Solution for eqn. 2

2.6.3. Solving 2st order ODE using 4th order Runge Kutta Method

The following program solves the following second order ODE from t=0 to t=5 using 50 steps:

$$\frac{d^2x}{dt^2}(t)+4\frac{dx}{dt}(t)+5x(t)=0 \quad x(0)=3,\frac{dx}{dt}(0)=-5 \quad \text{(eqn. 3)}$$

Then the numerical solution, the exact solution and the percentage error are stored in a matrix. The matrix are displayed on the screen and saved in a file named MoutputRK4.mat.

For obtaining numerical solution, eqn. 3 is rewritten to the following form:

$$\frac{dx}{dt}=y, x(0)=3$$

$$\frac{dy}{dt}=-5x-4y, y(0)=-5$$

```
function float output=dy(float x, float y)
        %differential equation
        output=-5*x-4*y;
end

function float output=dx(float x, float y)
        output=y;
end

function float output=x(float t)
        %Exact solution of dx
        output=3*exp(-2*t)*cos(t)+exp(-2*t)*sin(t);
end

function matrix E=RK4_2(float a, float b, float ya, float xa, int N)
%solves the diff eq using 4th order runge kutta method
%a is the initial value of the independent variable
%b is the end value of the indepenedent variable
%ya is the initial value of dy/dt
%xa is the initial value of dx/dt

%N is the number of steps
%output is a matrix with independent variable in the first column
and dependent variable in the second column

        float h;
        float g1;
        float g2;
        float g3;
        float g4;
        float f1;
        float f2;
        float f3;
        float f4;
        float y0;
        float y;
        int i;
```

```
        E=newmat(N+1,2);
        h=(b-a)/N;
        E[0,0]=a;
        E[0,1]=xa;

        for i=1:N
                E[i,0]=E[i-1,0]+h;
        end

        y0=ya;
        disp(y);
        for i=0:N-1
                g1=dy(E[i,1],y0);
                f1=dx(E[i,1],y0);

                g2=dy(E[i,1]+h/2*f1, y0+h*g1/2.);
                f2=dx(E[i,1]+h/2*f1, y0+h/2.0*g1);

                g3=dy(E[i,1]+h/2*f2, y0+h*g2/2.);
                f3=dx(E[i,1]+h/2*f2, y0+h/2.0*g2);

                g4=dy(E[i,1]+h*f3,y0+h*g3);
                f4=dx(E[i,1]+h*f3, y0+h*g3);

                y=y0+h*(g1+2*g2+2*g3+g4)/6.0;

                E[i+1,1]=E[i,1]+h*(f1+2*f2+2*f3+f4)/6.0;
                y0=y;
        end

end

function main()
        %this function solves the differential equation dy using 4th
        %order Runge Kutta method and displays the RK4 solution
        %with the exact solution and percentage error

        matrix R; %Solution using RK4 method
```

```
matrix Moutput; %output matrix
float Rexact; %exact solution
int n;
int i;

R=RK4_2(0.0, 5.0, -5.0, 3.0, 50);
setFormat("short");
n=height(R);
disp(n);
Moutput=newmat(n,4);
for i=0:n-1
        Rexact=x(R[i,0]);
        Moutput[i,0]=R[i,0]; %t
        Moutput[i,1]=R[i,1]; %Euler method solution
        Moutput[i,2]=Rexact; %exact solution
        Moutput[i,3]=(Moutput[i,2]-
  Moutput[i,1])/Moutput[i,2]*100.0; % percent error
end
disp("      t          RK4       Exact      %Error");
disp(Moutput);
save(Moutput,"MoutputRK4_2");

end
```

Program 2.6.3

Output:

t	RK4	Exact	%Error
0.0000	3.0000	3.0000	0.0000
0.1000	2.5256	2.5257	0.0005
0.2000	2.1040	2.1040	0.0009
0.3000	1.7351	1.7351	0.0012
0.4000	1.4165	1.4166	0.0015
0.5000	1.1449	1.1449	0.0017
0.6000	0.9158	0.9158	0.0018
0.7000	0.7247	0.7247	0.0018
0.8000	0.5668	0.5668	0.0017
0.9000	0.4377	0.4377	0.0015
1.0000	0.3332	0.3332	0.0011

1.1000	0.2495	0.2495	0.0004
1.2000	0.1832	0.1832	-0.0005
1.3000	0.1312	0.1312	-0.0019
1.4000	0.0909	0.0909	-0.0040
1.5000	0.0602	0.0602	-0.0073
1.6000	0.0372	0.0372	-0.0132
1.7000	0.0202	0.0202	-0.0254
1.8000	0.0080	0.0080	-0.0646
1.9000	-0.0005	-0.0005	0.9535
2.0000	-0.0062	-0.0062	0.0769
2.1000	-0.0098	-0.0098	0.0456
2.2000	-0.0117	-0.0117	0.0347
2.3000	-0.0126	-0.0126	0.0291
2.4000	-0.0126	-0.0126	0.0258
2.5000	-0.0122	-0.0122	0.0235
2.6000	-0.0113	-0.0113	0.0219
2.7000	-0.0103	-0.0103	0.0206
2.8000	-0.0092	-0.0092	0.0195
2.9000	-0.0081	-0.0081	0.0186
3.0000	-0.0070	-0.0070	0.0178
3.1000	-0.0060	-0.0060	0.0170
3.2000	-0.0051	-0.0051	0.0162
3.3000	-0.0042	-0.0042	0.0154
3.4000	-0.0035	-0.0035	0.0146
3.5000	-0.0029	-0.0029	0.0138
3.6000	-0.0023	-0.0023	0.0128
3.7000	-0.0019	-0.0019	0.0117
3.8000	-0.0015	-0.0015	0.0105
3.9000	-0.0012	-0.0012	0.0091
4.0000	-0.0009	-0.0009	0.0074
4.1000	-0.0007	-0.0007	0.0054
4.2000	-0.0005	-0.0005	0.0029
4.3000	-0.0004	-0.0004	-0.0002
4.4000	-0.0003	-0.0003	-0.0043
4.5000	-0.0002	-0.0002	-0.0099
4.6000	-0.0001	-0.0001	-0.0181
4.7000	-0.0001	-0.0001	-0.0311
4.8000	-0.0000	-0.0000	-0.0551

4.9000	-0.0000	-0.0000	-0.1148
5.0000	-0.0000	-0.0000	-0.5228

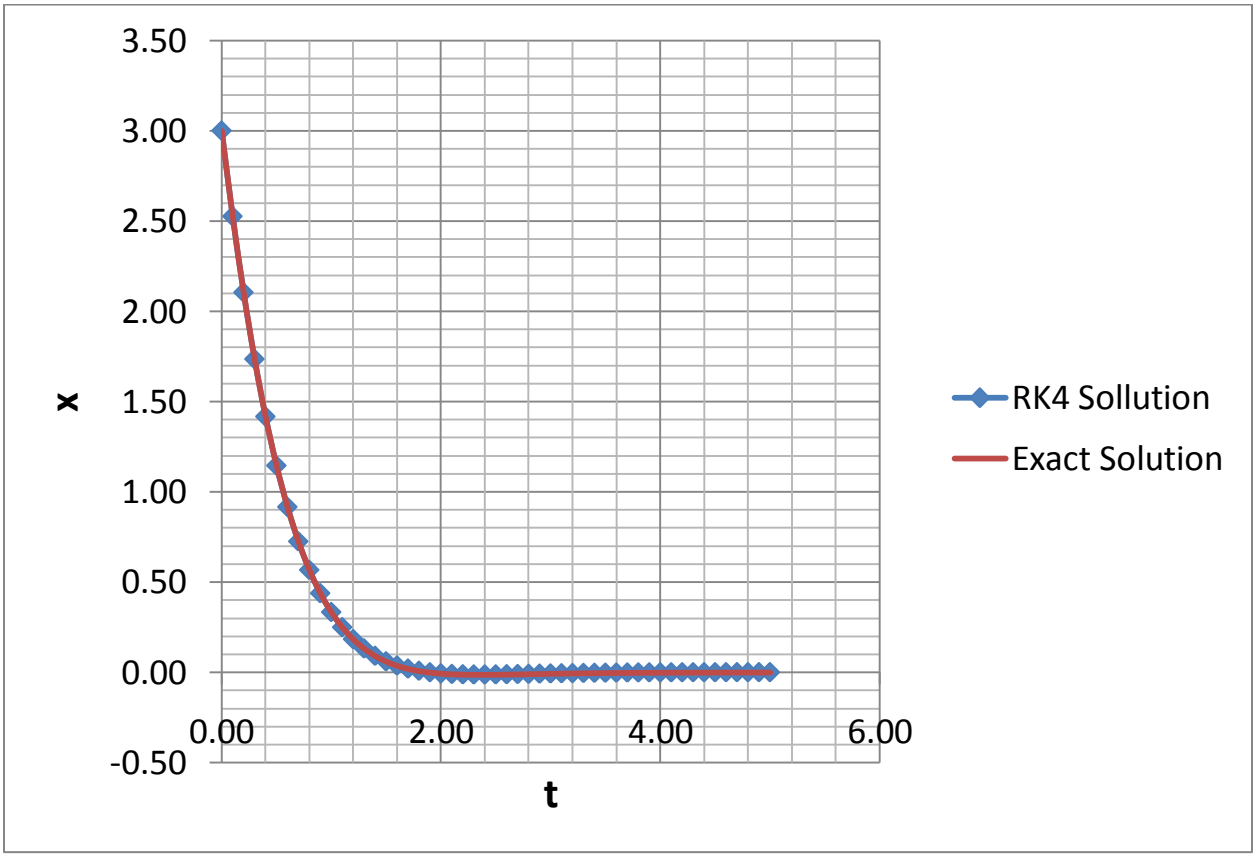


Figure 2: Comparison of RK4 Solution to Exact Solution for eqn. 3

The text output and Figure 2 show that the MSCL program worked as intended and the RK4 solution almost exactly match the exact analytical solution.

III. Language Reference Manual

1. Introduction

MSCL is a computer language designed to perform basic matrix operations and scientific computing. It provides a subset of MATLAB's matrix and mathematics capability. The syntax of MSCL is also similar to that of MATLAB. A program written in MSCL is first translated to a JAVA program and then compiled to run on Java Virtual Machine (JVM).

2. Lexical Conventions

There are six kinds of tokens: identifiers, keywords, numbers, strings, expression operators, and other special symbols. White spaces, tabs, newlines and comments are used to separate tokens, and they are ignored in other situations. At least one white space character is required to separate adjacent identifiers and constants.

2.1. Comments

The start of a comment is marked by the character "%" The end of a comment is marked by a newline character. Anything between the "%" and newline characters will be ignored by the compiler. Here is an example of a comment:

%this is a comment

2.2. Identifiers

An identifier is a sequence characters consisting of alphabets and digits. The first character must be an alphabet. The underscore "_" character counts as an alphabet it cannot be used as the first character in an identifier. Upper and lower case alphabets are considered different. There is no limit to the length of identifiers.

2.3. Reserved Keywords

The following is a list of reserved keywords that cannot be used for other purposes:

boolean	Break	continue	Else	Elseif
End	false	float	for	function
If	int	matrix	newmat	string
True	While			

2.4. Numbers

There are two types of numbers: integer numbers and and floating point numbers.

2.4.1. Integer number

An integer number is a sequence of digits from 0 to 9. All integers in MSCL are taken to be 32-bit integers.

2.4.2. Floating Point Number

A Floating point number in MSCL is defined the same way as that in the C reference manual: "A floating constant consists of an integer part, a decimal point, a fraction part, an e, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing." All floating point numbers are taken to be 64-bit double precision floating point numbers.

2.5. Strings

A string is a sequence of surrounded by double quotes “”.

2.6. Special Symbols

The following special symbols are used in the language for various purposes such as expression mathematical and logical operations, indicating comments and creating matrices.

+	-	*	.*	/	./	\	.\	^	=	==	<
>	<=	>=	!=	|	&	!	(	)	[	]	,
:	%	;									

3. Syntax Notation

In this manual, syntactic notations are indicated by *italic* type, and literal words and characters in gothic. Alternatives are listed on separate lines. An optional terminal or nonterminal symbol is indicated by the subscript ‘‘opt,’’ so that

[expression $_{opt}$ *]*

would indicate an optional expression in brackets. Subscript after an *expression* such as *expression* $_{1}$ is used to distinguish one *expression* in a sequence of *expressions*.

4. Types

There are five data types in MSCL: boolean(boolean), integer(int), double precision floating point numbers (float), string(string), and matrix(matrix). Because MSCL is translated to Java, all four data types follow specifications of the corresponding data types in Java.

4.1. boolean

boolean type has two possible values: true and false.

4.2. int

The int data type is a 32-bit signed two's complement integer. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive).

4.3. float

The `float` data type is a double-precision 64-bit IEEE 754 floating point number. It follows the same specification as the double data type in Java.

4.4. string

The string data type represents sequence of character and follows the same specification that of the String class in Java.

4.5. matrix

The matrix data type represents a m x n matrix. All elements in the matrix have type float. This data type is implemented in Java using the JAMA package developed by MathWorks and NIST.

4.6. Conversions

Because most scientific computing are done using floating point numbers, all mathematical operations in MSCL will convert int operands to float operands. In addition, when an int is assigned to a matrix element, it is converted to float.

5. Expressions

The following sections list the expressions and expression operators used in MSCL. The expression operators are listed in order of precedence (highest precedence first). For example, the expression operator in section 4.3 has higher precedence than that in section 4.4. Within each subsection, the operators have the same precedence. Left- or right-associativity of the operators are also specified.

5.1. Primary Expressions

5.1.1. *Identifier*

An identifier is bounded to an assigned value. The data type of the identifier depends on the value of the right-hand side of the assignment operator.

5.1.2. *Numbers*

A number is a constant and is evaluated to itself.

5.1.3. *Strings*

A sequence of characters enclosed in double quotes.

5.1.4. (*expression*)

A parenthesized expression evaluates to the same value as the expression enclosed by the parenthesis.

5.1.5. *identifier [expression$_1$, expression$_2$]*

An *identifier* of type matrix followed by two *expressions* separated by a comma and enclosed in brackets denotes accessing the matrix element at row = *expression$_1$* ***and column*** = *expression$_2$*. *expression$_1$* ***and*** *expression$_2$* should be int type. Matrix access start with index 0.

5.1.6. *identifier [expression$_1$: expression$_2$, expression$_3$:expression$_4$]*

This primary expression is used to access a sub-matrix from initial row *expression*$_1$ to final row *expression*$_2$, from initial column *expression*$_3$ to final column *expression*$_4$

5.1.7. *identifier (expression-list*$_{opt}$)

A function call is a primary expression followed by an optional list of expressions separated by commas enclosed in parentheses which constitutes the actual argument to the function. All arguments are passed by value.

5.2. Unary Operators

Expressions with unary operators group right-to-left

5.2.1. *– expression*

The result is negative of *expression. expression* must be type int or float.

5.2.2. *! expression*

The result is the logical negation of *expression. expression* must be type boolean.

5.3. Matrix transpose operator: *expression '*

The result is the transpose of matrix *expression*.

5.4. Power Operator*: expression*$_1$^ *expression*$_2$

The result is ***expression***$_1$ to the power of ***expression***$_2$. This operator groups right-to-left. *expression* can be type int or float. The result is converted to float.

5.5. Multiplicative Operators

Multiplicative operators *, .*, /, ./, \, and .\ group left-to-right

5.5.1. *expression * expression*

The binary operator * indicates multiplication. If both expressions are type int, the result is type int. If both expressions are type float, the result is type float. If one expression is type float and the other is type int, the result is type float.

If one expression if type matrix and the other is type int or float, the result is a matrix in which each element is multiplied by the int or float expression.

5.5.2. *expression .* expression*

The binary operator .* indicates matrix element-by-element multiplication of two expressions of type matrix.

5.5.3. *expression* / *expression*

The binary operator / indicates division. If both expressions are type int, the result is type int. If both expressions are type float, the result is type float. If one expression is type float and the other is type int, the result is type float. If one expression if type matrix and the other is type int or float, the result is a matrix in which each element is multiplied by the int or float expression.

5.5.4. *expression* ./ *expression*

The binary operator ./ indicates matrix element-by-element right division of two expressions of type matrix.

5.5.5. *expression* \ *expression*

The binary operator \ indicates matrix left division of two expressions of type matrix. The A\b is equal to x in Ax=b.

5.5.6. *expression* .\ *expression*

The binary operator .\ indicates matrix element-by-element left division of two expressions of type matrix.

5.6. Additive Operators:

Additive operators + and – group left to right.

5.6.1. *expression* + *expression*

The result is sum of expressions. If both expressions are type int, the result is type int. If both expressions are type float, the result is type float. If one expression is type float and the other is type int, the result is type float.
If one expression if type matrix and the other is type int or float, the result is a matrix in which each element is multiplied by the int or float expression.
If both expressions are type matrix then the result is matrix addition.

5.6.2. *expression* – *expression*

The result is difference of expressions. If both expressions are type int, the result is type int. If both expressions are type float, the result is type float. If one expression is type float and the other is type int, the result is type float.
If one expression if type matrix and the other is type int or float, the result is a matrix in which each element is multiplied by the int or float expression.

If both expressions are type matrix then the result is matrix subtraction.

5.7. Relational Operators

The relational operators listed below group left-to-right. All relational operators yield true if the relation is true, and false if the relation is false;

5.7.1. *expression < expression:* less than

5.7.2. *expression > expression:* greater than

5.7.3. *expression <= expression*: less than or equal to

5.7.4. *expression >= expression:* greater than or equal to

5.8. Equality Operators

The equality operators behave the same as relational operators. But they have lower precedence.

5.8.1. *expression == expression:* equal to

5.8.2. *expression != expression:* not equal to

5.9. Logical *not* operator: ! *expression*

The ! operator takes boolean expressions as operands and it groups left-to-right. The result is the logical negation of the operands

5.10. Logical *and* operator: *expression & expression*

The & operator takes two boolean expressions as operands and it groups left-to-right. The result is the logical *and* of the two operands. This operator is short-circuited.

5.11. Logical *or* operator: *expression | expression*

The | operators takes two boolean expressions as operands and it groups left-to-right. The result is the logical *or* of the two operands. The operator is short-circuited.

5.12. Assignment operator: *identifier = expression*

This expression replaces the value of *identifier* with the value of *expression.* The types of *identifier* and *expression* must be the same.

6. Declarations

Declarations are used to specify the type of identifiers. Declaration of local variables within a function must be done at the beginning of the function. Declaration of global variables can be done in any order outside of function definitions. The following declarations are possible:

int integer;
float floating;
boolean bool;
matrix M;
After declaration, an initial value is assigned to each identifier. The following are the default initial values of the above declarations:
int : 0
float : = 0.0
boolean: false
matrix: 1x1 matrix with value 0.0

7. Statements

If not otherwise indicated, statements are executed in sequence.

7.1. Expression statement: *statement*

Most statements are expression statements which has the form
expression ;
The semicolon is used as a statement separator.

7.2. Statement list: *statement-list*

A statement list is a list of one or more statements separated by ";". It has the following form:
statement-list:
statement
statement statement-list

7.3. Matrix value initialization

The element values of a *matrix* can be initialized using the following statement:
Identifier = [element_list];
element-list:
| expression
| element-list , expression
| element-list ; expression
For example the matrix:

$$M = \begin{vmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{vmatrix}$$

is initialized using the following statement: *M=[0, 1, 2 ; 3, 4, 5];*

7.4. Conditional statement

The possible formats of the conditional statements are listed in the following sections.

7.4.1. *if-else*

if (expression)

statemen- list

else

statement-list$_{opt}$

end

expression is a logical expression. If the logical expression after *if* is true, then the first *statement-list* is executed. Otherwise, the second optional *statement-list* after *else* is executed. A semicolon should not be placed after *end*. The statement after else is optional. However, the else is not optional.

7.4.2. *If-elseif-else*

if (expression)

statement-list

elseif (expression)

statement-list

.

.

else

statement-list$_{opt}$

end

expression is logical expression. If the logical expression after *if* is true, then the first *statement-list* is executed. Following the *if* statement, there can be any number of *elseif* statements. If the logical expression in the *if* statement is false, then the *statement-list* inside the first logical expression after *elseif* that evaluate to true will be executed. If the logical expressions after both “if” and “elseif” statements return false, then the optional *statement-list* under *else* will be executed. A semicolon should not be placed at after *end*. The statement after else is optional. However, the else is not optional.

7.5. *For* statement

The *for* statement is an iterative statement in the following format:

for for_expression

statement-list

end

for_expression:

| *identifier = expression$_1$: expression$_2$*
| *identifier = expression$_1$: expression$_2$: expression$_3$*

If the *for_expression* is in the form *identifier = expression$_1$: expression$_2$, expression$_1$* is the start value of *identifier*. The value *identifier* is incremented by *1* and the *statement-list* is executed in each iteration. The loop is exited whe *identifier is greater than expression$_2$* at the start of a iteration.
If the *for_expression* is in the form *identifier = expression$_1$: expression$_2$: expression$_3$. expression$_1$* is the start value of *identifier*. The value *identifier* is incremented by *expression$_2$* and the *statement-list* is executed in each iteration. The loop is exited whe *identifier is greater than expression$_3$* at the start of a iteration. Semicolon should not be used after *end*.

7.6. While statement

This statement is an iterative statement in the following format:
while (expression)
statemen- list
end
The *statemen-list* is executed repeatedly as long as the value of the logical *expression* remains *true*. The test takes place before each execution of the statement list. Semicolon should not be used after *end*.

7.7. *break* statement

The statement
break;
is used inside a *for* or *while* statements to cause the termination of the innermost *for* or *while* iterative statements. Control is passed to the statement following the terminated statement.

7.8. *continue* statement

The statement:
continue;
is used inside a *for* or *while* to cause the current iteration of the innermost iterative statement to terminate and proceed to the next iteration.

8. Function definition

A function can be defined in two ways depending on its return type. If the function does not return anything, the following form is used:
function identifier (parameter-list$_{opt}$)

var-declare-list
statement-list
end

parameter-list$_{opt}$:
| */* nothing */*
| *formal_list*
formal_list:
| *param_decl*
| *formal_list , param_decl*
param_decl:
| *datatype ID*
var-declare_list:
| */* nothing */*
| *var-declare_list var-declare*

var-declare:
datatype identifier;

For example, the *main()* function is declared using the following statement:
function main()
int a;
a=1;
end
if the function returns a value, the following form is used:
function datatype identifier = identifier (parameter-list$_{opt}$)
var-declare-list
statement-list
end

For example, the following function returns the sums of the two parameters:
function int a = foo(int x, int y)
a=x+y;
end

Each program must have a *main()* function where program execution starts. Functions can be declared in any order.

9. Build-in functions

9.1. *disp (expression)*

This function displays the value of the *expression* on screen.

9.2. *save (expression, string)*

This function saves the value of matrix *expression* to a comma separated ASCII file with name *string* in the current directory.

9.3. *load (string)*

This function loads a value of matrix stored in a file with name *string*. This file is saved by the *save* function described in section 9.2;

9.4. *tostring (expression)*

This function converts *int*, *boolean or float expression* into a *string expression*.

9.5. *setFormat (string)*

This function sets the format of the numerical outputs of the *disp* function. The following value for *string* is possible:

"*short":* floating point number 4 decimal places
"*shortE":* scientific notation with 4 decimal places
"*long":* floating point number 10 decimal places
"*longE":* scientific notation with 10 decimal places

9.6. Matrix functions

9.6.1. *newmat(expression$_1$, expression$_2$)*

This function returns a matrix with dimension *expression$_1$* by *expression$_2$* *where expression$_1$ and expression$_2$* are of type *int*

9.6.2. *width(expression)*

This function returns the number of columns in *matrix expression* as an *int*.

9.6.3. *height(expression)*

This function returns the number of rows in *matrix expression* as an *int*.

9.6.4. *inv (expression)*

This function returns the inverse of *matrix expression*.

9.7. Mathematical functions

The following lists the build-in mathematical functions of MSCL. All functions take *int* or *float* expressions as argument and return values of type *float*.

abs(*expression*) absolute value of *expression*

sqrt(*expression)*	square root of of expression
exp(*expression*)	$e^{expression}$
log(*expression*)	natural log of *expression*
log10(*expression*)	log to the base 10 of *expression*
sin(*expression*)	sine of *expression*
cos(*expression*)	cosine of *expression*
tan(*expression*)	tangent of *expression*
asin(*expression*)	arc sine of *expression*
acos(*expression*)	arce cosine of *expression*
atan(*expression*)	arc tangent of *expression, return value between* $-\frac{\pi}{2}$ *and* $\frac{\pi}{2}$
atan2(*expression*$_1$, *expression*$_2$)	arc tangent of *expression*$_2$/ *expression*$_1$, return value bewteen $-\pi$ *and* π
mod(*expression*$_1$, *expression*$_2$)	modular division of *expression*$_2$ by *expression*$_1$

10. Scope rules

MSCL uses static scoping. There are two types of variable scopes: global and local. A global variable is a variable that is assigned a value outside of a function definition. Global variables can be used anywhere inside a program except when there is a local variable of the same name in a function. Global variables exist until the termination of the program. A local variable is a variable that is assigned a value inside a function definition. Local variables can only be used inside the function in which it was first assigned a value, and it cannot be used outside of that function. If there is a global variable and a local variable with the same name inside a function, the value of the local variable will be used instead of the global variable.

IV. Project Plan

1. Development Process

The initial plan for this project was to develop a MATLAB-like language that is capable manipulating matrices and solving ordinary differential equations from scratch using OCaml. However, it soon became obvious that this is not possible given that I am doing the project by myself, and a demanding work schedule at my job. Therefore, I decided to create my compiler by modifying another existing compiler – MATLIP that was developed by Pin-Chin Huang et.al. for a previous class project. MATLIP was chosen because it already used syntax similar to that of

MATLAB. Since I know how to program in Java, the fact that MATLIP translates source codes into Java codes is also an advantage.

In developing MSCL, several new features are added to MATLIP. For example, direct matrix manipulation capabilities, various trigonometric and other math functions, continue and break statements are new feature in MSCL. In addition, MSCL allows the mixing of integer, float and matrix operands in a math expression to make constructing calculations easier. A bug in the original MATLIP program that omits parenthesis in math expressions has been fixed. Finally, several new test cases for solving differential equations are developed for MSCL.

2. Software Development Environment

The compiler is developed using a combination of OCaml and Netbeans. Since MSCL source codes are translated to Java, codes for matrix manipulation and I/O are first developed in Netbeans and then cut and pasted into OCaml source code for the MSCL compiler. All source codes are developed in Windows XP operating system.

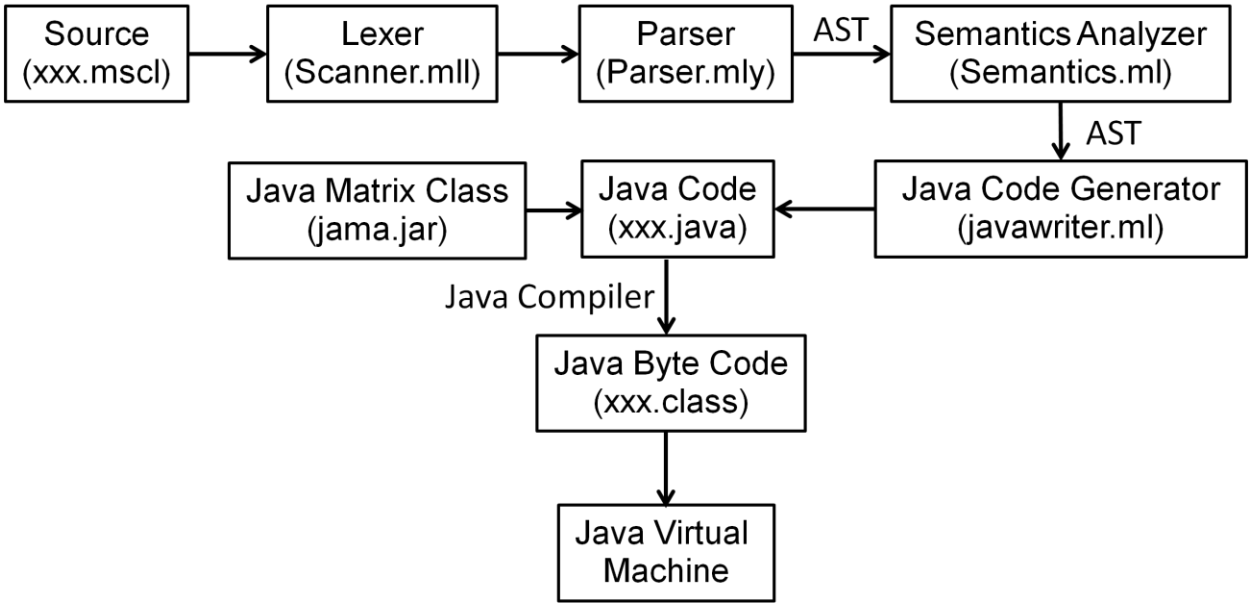
3. Project Schedule

09/8 – 10/15	• Learning OCaml • Develop ideas for the project • Research previous projects
10/16 – 11/03	• Developed a preliminary scanner and parser • Completed preliminary language reference manual • Continue to learn OCaml using previous projects' source codes
11/04 – 12/13	• Complete development of scanner, parser, semantic analyzer, and java source code generator • Complete develop of Java codes for matrix manipulation using JAMA.
12/14 – 12/22	• Complete testing • Complete final project report

V. Architectural Design

The MSCL compiler consists of the lexer, the parser, the semantics analyzer, and the java code generator, all of which are written in Ocaml. The lexer takes a MSCL source code (a *.mscl file) as input and outputs a stream of tokens to parser. The Parser parses those tokens and builds an unambiguous abstract syntax tree (AST). Then the semantics analyzer analyzes the syntax tree to ensure the program is semantically correct. It builds three symbol tables for: local variables, global variables and functions.

It also performs type checking. If no error are found by the semantics analyzer, the Java code generator walks through the AST again and translate the MSCL source code into Java code. The java code generated by MSCL compiler imports the open-source Java Matrix Package (JAMA.jar) developed by MathWorks and NIST. Finally the Java compiler compiles the Java source into byte code to be run on Java Virtual Machine.



VI. Test Plan

Tests are performed as new features are added to MSCL. After all major features of MSCL have been implemented, a series of test cases are developed for regression testing. Test cases are split into five categories: math operations, matrix operations, control flow, scope rules, and solution of ODE's. The following lists all the test cases that are included in the project submission:

1. Test Cases

1.1. Testing for Math Operations

Test cases for math operations are constructed by chaining various operations and math function. An example of this testing category is Program 2.3 on page 6. The following are the files used for testing math operation:

test_math1.mscl (simple addition)
test_math2.mscl (simple addition with negative numbers)
test_math3.mscl (floating point operations with exponential function)
test_math4.mscl (chaining of operations with parenthesis)

test_math5.mscl (complex chain of operations with trigonometric functions)
test_math6.mscl (trigonometric functions)
test_minus.mscl (chaining of several unary minus operations)

1.2. Testing for Matrix Operations

Test cases for matrix operations are constructed to check binary operations such as matrix addition, subtraction, multiplication and division can be performed. In addition, matrix assignments, access, inverse, transpose, save and load are also tested. An example of test case in this category is Program 2.6.1 on page 10.

test_matrix_add.mscl (matrix addition)
test_matrix_assign.mscl (matrix assignment within a loop)
test_matrix_dim_inv.mscl (get matrix dimension and inverse)
test_matrix_div.mscl (matrix and scalar division)
test_matrix_elementOp.mscl (element-by-element matrix operation)
test_matrix_mult.mscl (matrix multiplication)
test_matrix_range.mscl (matrix range access)
test_matrix_rotate.mscl (matrix multiplication for vector rotation)
test_matrix_save_load.mscl (matrix save and load)
test_matrix_transpose.mscl (matrix transpose)

1.3. Testing for Control Flow

Test cases for control flow are constructed to ensure that nested *for*, *while* and *if* statements work as intended. Regular function calls, recursive function calls and calling functions in function parameters are also tested. The following test cases are adopted from MATLIP's test suite. An example of this category of test case is Program 2.4.4.

test_for_intsum.mscl (for loop with integer loop variable)
test_for_nested.mscl (nested for loop with integer loop variables)
test_for_nested2.mscl (nested for loop with integer loop variables)
test_for_nested3.mscl (nested for loop with float loop variables)
test_for_nested4.mscl (nested for loop with float loop variables and negative step size)
test_for_while.mscl (while loop)
test_fun.mscl (function call)
test_if1.mscl (if…else…end)
test_if2.mscl (nested if…else…end)

test_if3.mscl	(nested if…else…end)
test_if4.mscl	(if…elseif…else…end)
test_if5.mscl	(nested if statements)
test_if6.mscl	(nested if statements)
test_nestedfun.mscl	(function call within function parameters)
test_recfun.mscl	(recursive function)

1.4. Testing for Control Flow

The following test cases are constructed to ensure the scoping rules specified in the language reference manual work as intended.

test_namespace1.mscl	(global variable name is the same with global function name)
test_namespace2.mscl	(local variable is the parameter of the global function)
test_namespace3.mscl	(local variable is inside the scope of the global function)
test_namespace4.mscl	(local variable is inside the scope of the global function)
test_namespace5.mscl	(set global variable value)
test_namespace6.mscl	(local variable has the same name at the global variable)

1.5. Testing for ODE Solutions

Test cases for solving ODE's are constructed to ensure that MSCL can handle more complicated numerical algorithms. This category of tests is the most advanced testing applied to MSCL. Program 2.6.2 and 2.6.3 are examples of this category of testing.

odeEuler.mscl	(1st order ODE using Euler's method)
odeHeun.mscl	(1st order ODE using Heun's method)
odeRK4.mscl	(1st order ODE using Rk4 method)
odeRK4_2.mscl	(2nd order ODE using RK4 mehtod)
odeRK4_3.mscl	(2nd order ODE using Rk4 method)

2. Automated Testing

After the results of the MSCL programs shown in the previous sections have been validated, the output of each program is written to a file with extension *.check. The *.check files are used as the validated results. For automated regressive testing, a batch file is run to execute all the test cases. The batch file outputs the result of each

program to a file with extension *.out. To validate the compiler, the *.out file are compared to the corresponding *.check using the command ‘‘comp *.check *.out’’. The following is a partial output of the automated test script:

```
Comparing test_math1.check and test_math1.out...
Files compare OK

Comparing test_math2.check and test_math2.out...
Files compare OK

Comparing test_math3.check and test_math3.out...
Files compare OK

Comparing test_math4.check and test_math4.out...
Files compare OK

Comparing test_math5.check and test_math5.out...
Files compare OK

Comparing test_math6.check and test_math6.out...
Files compare OK
```

“Files compare OK” indicates the output file is identical to the correct output stored in the corresponding *.check file.

VII. Lessons Learned

I think the OCaml source codes and final report from previous classes were very helpful. Before starting the project, I searched through all the previous project reports for inspiration and to see what things are possible with OCaml. When I did not know how to do something in OCaml, I was able to find the solution in other people’s source code. I was fortunate to have access to MATLIP’s source code. Otherwise, if I had to start from scratch, I may not have completed everything I wanted to accomplish in this project.

VIII. References

1. JAMA: A Java Matrix Package, developed by MathWorks and NIST, http://math.nist.gov/javanumerics/jama/

2. Pin-Chin Huan et.al., “MATLIP: MATLAB-Like Language for Image Processing Final Report”, 12/19/2008.

IX. Complete Listing

Ast.mli

```
type op = Add | Sub | Mul | Div | Equal | Neq | Less | Leq | Greater | Geq | And |
Or | Power | Emul | Erdiv | Ldiv | Eldiv

type expr =
    Literal of int
  | Floatlit of float
  | Bool of string
  | Paren of expr
  | Id of string
  | Datatype of string
  | String of string
  | Binop of expr * op * expr
  | Transpose of expr
  | Uminus of expr
  | Not of expr
  | Assign of string * expr
  | Call of string * expr list
  | AssignMatrix of string*expr list*expr
  | MatrixAccess of string*expr list
  | MatrixRAccess of string*expr*expr*expr*expr
  | Newmat of expr*expr


type for_expr =
    Assigna of string * expr * expr
  | Assignb of string * expr * expr * expr

type stmt =
    Block of stmt list
  | Expr of expr
  | Break
  | Continue
  | MatrixInit of string * expr list list
  | If of expr * stmt * stmt
```

```
  | Ifelseif of expr * stmt * elseif list * stmt
  | For of for_expr * stmt
  | While of expr * stmt
and elseif = {
   elseif_expr : expr;
   elseif_stmt : stmt;
 }

type var_decl = {
   varname : string;
   vartype : string;
 }

type func_decl = {
   fname : string;
   rettype : string;
   retname : string;
   formals : var_decl list;
   locals : var_decl list;
   body : stmt list;
 }

type program = var_decl list * func_decl list
```

Scanner.mll

```
{
   open Parser
   (*Increment the line number tracked by lexbuf*)
   let incrLineNum lexbuf =
      let pos = lexbuf.Lexing.lex_curr_p in
         lexbuf.Lexing.lex_curr_p <- { pos with
            Lexing.pos_lnum = pos.Lexing.pos_lnum + 1;
            Lexing.pos_bol = pos.Lexing.pos_cnum;
         }

}

let floatLit = ['0'-'9']+['.']['0'-'9']* (['e''E'](['-''+']?['0'-'9']+))?
| ['0'-'9']*['.']['0'-'9']+ (['e''E'](['-''+']?['0'-'9']+))?
| ['0'-'9']*['.']?['0'-'9']+ (['e''E']['-''+']?['0'-'9']+)
```

```
rule token = parse
  [' ' '\t']     { token lexbuf }
 | ['\r' '\n']    { incrLineNum lexbuf; token lexbuf }
 | '%'            { comment lexbuf }
 | ""             { stringParse "" lexbuf}
 | '('            { LPAREN }
 | ')'            { RPAREN }
 | '['            { LBRACKET }
 | ']'            { RBRACKET }
 | ';'            { SEMI }
 | ','            { COMMA }
 | ':'            { COLON }
 | '^'            { POWER }
 | '*'            { MUL }
 | ".*"           { EMUL}
 | "./"           { ERDIV}  (*element by element right divid*)
 | '\\'           { LDIV}   (*matrix left divide*)
 | ".\\"          {ELDIV}  (*element by element left divide*)
 | '/'            { DIVIDE } (*divide or matrix right divide*)
 | '\''           {TRANSPOSE}
 | '+'            { PLUS }
 | '-'            { MINUS }
 | '='            { ASSIGN }
 | "=="           { EQ }
 | "!="           { NOTEQ }
 | '<'            { LT }
 | "<="           { LTEQ }
 | ">"            { GT }
 | ">="           { GTEQ }
 | "&"            { AND }
 | "|"            { OR }
 | "!"            { NOT }
 | "if"           { IF }
 | "else"         { ELSE }
 | "elseif"       { ELSEIF }
 | "true"  as lxm { BOOL(lxm) }
 | "false" as lxm { BOOL(lxm) }
 | "for"          { FOR }
 | "while"        { WHILE }
 | "continue"     { CONTINUE }
 | "break"        { BREAK }
```

```
| "end"         { END }
| "function"    { FUNCTION }
| "boolean"   { DATATYPE("boolean") }
| "int"  { DATATYPE("int") }
| "float"         { DATATYPE("float") }
| "string"        { DATATYPE("string") }
| "matrix"       { DATATYPE("matrix") }
| "newmat"     { NEWMAT }
| ['0'-'9']+ as lxm  { LITERAL(int_of_string lxm) }
| floatLit as lxm    { FLOATLIT(float_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char {
          let pos = lexbuf.Lexing.lex_curr_p in
          raise (Failure("Illegal character: " ^ Char.escaped char
          ^ " in line #" ^ (string_of_int pos.Lexing.pos_lnum))) }

and comment = parse
  "\n" { incrLineNum lexbuf; token lexbuf }
| _    { comment lexbuf }

and stringParse str = parse
| ""  { STRING(str) } (* end of a string literal *)
| [^ ""] as c  { stringParse (str ^ String.make 1 c) lexbuf } (*everything except quote is
counted as a character inside the string*)
|  "\n" {let currPos = lexbuf.Lexing.lex_curr_p in
          raise (Failure("Unclosed string literal, in line #" ^ (string_of_int
currPos.Lexing.pos_lnum))) }
|  eof  {let currPos = lexbuf.Lexing.lex_curr_p in
          raise (Failure("Unclosed string literal, in line #" ^ (string_of_int
currPos.Lexing.pos_lnum))) }
```

Parser.mly

```
%{  open Ast
    open Lexing
    let parse_error msg =
        let start_pos = Parsing.rhs_start_pos 1 in
                                        let lineNo = start_pos.pos_lnum in
                                        print_endline (msg ^ " in line #" ^ string_of_int lineNo)
%}
```

```
%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA COLON LBRACKET RBRACKET NEWMAT
%token PLUS MINUS MUL DIVIDE ASSIGN  POWER EMUL ERDIV LDIV ELDIV TRANSPOSE
%token EQ NOTEQ LT LTEQ GT GTEQ AND OR NOT
%token IF ELSE ELSEIF FOR WHILE BREAK CONTINUE END FUNCTION
%token <int> LITERAL
%token <float> FLOATLIT
%token <string> ID
%token <string> DATATYPE
%token <string> STRING
%token <string> BOOL
%token EOF

%nonassoc NOELSE
%nonassoc ELSE

%nonassoc NOACTUALS
%nonassoc LPAREN

%nonassoc NOMINUS

%right ASSIGN
%left OR
%left AND
%left EQ NOTEQ
%left LT GT LTEQ GTEQ
%left PLUS MINUS
%left MUL DIVIDE EMUL LDIV ELDIV ERDIV
%right POWER
%left TRANSPOSE
%right NOT

%nonassoc UMINUS

%start program
%type <Ast.program> program

%%

program:
  /* nothing */ { [], [] }
```

```
 | program vdecl { ($2 :: fst $1), snd $1 }
 | program fdecl { fst $1, ($2 :: snd $1) }

fdecl:
   FUNCTION DATATYPE ID ASSIGN ID LPAREN formals_opt RPAREN vdecl_list stmt_list END
    { { fname = $5;
    rettype = $2;
    retname = $3;
          formals = $7;
          locals = List.rev $9;
          body = List.rev $10 } }
  |  FUNCTION ID LPAREN formals_opt RPAREN vdecl_list stmt_list END
    { { fname = $2;
    rettype = "void";
    retname = "";
          formals = $4;
          locals = List.rev $6;
          body = List.rev $7 } }

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

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

param_decl:
   DATATYPE ID
    { { varname = $2;
    vartype = $1 } }

vdecl_list:
    /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }

vdecl:
   DATATYPE ID SEMI
    { { varname = $2;
    vartype = $1 } }
```

```
stmt_list:
    /* nothing */ %prec NOMINUS { [] }
  | stmt_list stmt { $2 :: $1 }



stmt:
    expr SEMI { Expr($1) }
  | error SEMI { Expr(Datatype("parseerror")) }
  | BREAK SEMI { Break }
  | CONTINUE SEMI { Continue }
  | IF expr stmt_list ELSE stmt_list END
      { If($2, Block(List.rev $3), Block(List.rev $5)) }
  | IF expr stmt_list elseif_list ELSE stmt_list END
      { Ifelseif($2, Block(List.rev $3), List.rev $4, Block(List.rev $6)) }
  | FOR for_expr stmt_list END
      { For($2, Block(List.rev $3)) }
  | WHILE expr stmt_list END { While($2, Block(List.rev $3)) }
  | ID ASSIGN LBRACKET matrix RBRACKET SEMI
      { MatrixInit($1, List.rev $4) }

matrix: /*matrix definition*/
    expr         { [[$1]] }
  | matrix COMMA expr  {
      if (List.length $1) = 1 then
         [$3 :: List.hd $1]
      else
         ($3 :: List.hd $1) :: List.tl $1 }
  | matrix SEMI expr  { [$3] :: $1 }



expr:
    LITERAL          { Literal($1) }
  | FLOATLIT         { Floatlit($1) }
  | BOOL             { Bool($1) }
  | ID %prec NOACTUALS     { Id($1) }
  | STRING           { String($1) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr MUL  expr { Binop($1, Mul,  $3) }
```

```
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr AND   expr  { Binop($1, And,   $3) }
  | expr OR  expr    { Binop($1, Or,   $3) }
  | expr POWER  expr { Binop($1, Power,  $3) }
  | NOT expr         { Not($2) }
  | MINUS expr %prec UMINUS    { Uminus($2) }
  | expr TRANSPOSE {Transpose($1)}
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NOTEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LTEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater,  $3) }
  | expr GTEQ    expr { Binop($1, Geq,   $3) }
  | expr EMUL    expr {Binop($1, Emul,$3)}
  | expr ERDIV  expr {Binop($1, Erdiv, $3)}
  | expr LDIV expr {Binop($1, Ldiv, $3)}
  | expr ELDIV  expr {Binop($1, Eldiv, $3)}
  | ID ASSIGN expr   { Assign($1, $3) }
  | ID LBRACKET actuals_list RBRACKET ASSIGN expr
      {AssignMatrix($1, $3, $6)}
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { Paren($2) }
  | ID LBRACKET actuals_list RBRACKET
      { MatrixAccess($1, $3) }
  | ID LBRACKET expr COLON expr COMMA expr COLON expr RBRACKET {MatrixRAccess
($1, $3, $5, $7, $9)}
  | NEWMAT LPAREN expr COMMA expr RPAREN
      { Newmat($3, $5) }

for_expr:
   ID ASSIGN expr COLON expr %prec NOMINUS {Assigna($1, $3, $5)}
  | ID ASSIGN expr COLON expr COLON expr %prec NOMINUS
   {Assignb($1, $3, $5, $7)}

elseif_list:
   elseif_clause            { [$1] }
  | elseif_list elseif_clause { $2 :: $1 }

elseif_clause:
   ELSEIF expr stmt_list
    { {elseif_expr = $2;
```

```
      elseif_stmt = Block(List.rev $3)} }

actuals_opt:
    /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

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

Semantics.ml

```
open Ast

module NameMap = Map.Make(struct
  type t = string
  let compare x y = Pervasives.compare x y
end)

let loop_flag = ref false;;
let string_of_program (vars, funcs) =
  (* Put function declarations in a symbol table *)
  let func_decls = List.fold_left
      (fun funcs fdecl ->
         if NameMap.mem fdecl.fname funcs then
           raise (Failure ("Function: '" ^ fdecl.fname ^
             "' has already been defined."))
         else
           NameMap.add fdecl.fname fdecl funcs) NameMap.empty funcs
   in
   let blackhole e = () in

   let rec string_of_fdecl globals fdecl actuals =

   let rec string_of_expr globals locals fname = function
         Literal(l) -> { varname = string_of_int l;vartype = "int" }
     | Floatlit(f) -> { varname = string_of_float f;vartype = "float" }
     | String(s) -> { varname = s;vartype = "string" }
     | Bool(s) -> { varname = s;vartype = "boolean" }
     | Id(s) ->
         if NameMap.mem s locals then
```

```
            NameMap.find s locals
        else if NameMap.mem s globals then
            NameMap.find s globals
        else raise (Failure ("Undeclared identifier found in expression: "'
            ^ s ^ "' in function: "' ^ fname ^ """))
    | Datatype(t) ->
        if t = "parseerror" then
            raise ((Failure (""")))
        else { varname = "null" ;vartype = t }
    | Paren (e) -> string_of_expr globals locals fname e
    | Transpose(e)->let v = string_of_expr globals locals fname e in
        if v.vartype = "matrix" then
            v
        else
            raise (Failure ("transpose operator ' can only be applied to matrix expressions"
                    ^ ", but here used with "' ^ v.varname ^ "', type: "
                    ^ v.vartype ^ " in function: "' ^ fname ^ """))
    | Uminus(e) -> let v = string_of_expr globals locals fname e in
        if v.vartype = "int" || v.vartype = "float" then
            v
        else
            raise (Failure ("unary - operator can only be applied to int and float
expressions"
                    ^ ", but here used with "' ^ v.varname ^ "', type: "
                    ^ v.vartype ^ " in function: "' ^ fname ^ """))
    | Not(e) -> let v = string_of_expr globals locals fname e in
        if v.vartype = "boolean" then
            v
        else
            raise (Failure ("not operator can only be applied to boolean expression"
                    ^ ", but here used with "' ^ v.varname ^ "', type: "
                    ^ v.vartype ^ " in function: "' ^ fname ^ """))
    | Binop(e1, o, e2) ->
      let v1 = string_of_expr globals locals fname e1 in
      (match o with
        Add ->
        let v2 = string_of_expr globals locals fname e2 in
          if v1.vartype = "string" then
            if v2.vartype = "matrix" then
              raise (Failure ("Cannot concatenate " ^ v2.vartype ^ " type with string type "
^
```

```
                    "in function: "' ^ fname ^ '"'))
          else v1
        else if v2.vartype = "string" then
          if v1.vartype = "matrix" then
            raise (Failure ("Cannot concatenate " ^ v1.vartype ^ " type with string type "
^
                    "in function: "' ^ fname ^ '"'))
          else v2
        else if v1.vartype = "matrix" && v2.vartype = "int" then
              v1
        else if v1.vartype = "matrix" && v2.vartype = "float" then
              v1
        else if v1.vartype = "int" && v2.vartype = "matrix" then
              v2
        else if v1.vartype = "float" && v2.vartype = "matrix" then
              v2
        else if v1.vartype = "matrix" && v2.vartype = "matrix" then
              v1
        else if v1.vartype = "float" && v2.vartype="int" then
             v1
        else if v1.vartype = "int" && v2.vartype="float" then
             v2
        else if v1.vartype = "boolean" || v2.vartype = "boolean" then
          raise (Failure ("+ operator cannot be applied to boolean operands"
                    ^ ", but here used with "' ^ v1.varname ^ '", type: "
                    ^ v1.vartype ^ " and "' ^ v2.varname ^ '", type: "
                    ^ v2.vartype ^ " in function: "' ^ fname ^ '"'))
        else if v1.vartype <> v2.vartype then
          raise (Failure ("Type mismatch in binary operation between: "' ^
                    v1.varname ^ '", type: " ^ v1.vartype ^ " and "' ^ v2.varname
                    ^ '", type: " ^ v2.vartype ^ " in function: "' ^ fname
                    ^ '"'))
        else v1
      | Sub ->
      let v2 = string_of_expr globals locals fname e2 in
        if v1.vartype = "string" || v2.vartype = "string" then
          raise (Failure ("Operators '-' " ^
                    "cannot be applied to a string, in function: "' ^ fname ^ '"'))
        else if v1.vartype = "matrix" && v2.vartype = "int" then
              v1
        else if v1.vartype = "matrix" && v2.vartype = "float" then
```

```
        v1
    else if v1.vartype = "int" && v2.vartype = "matrix" then
        v2
    else if v1.vartype = "float" && v2.vartype = "matrix" then
        v2
    else if v1.vartype = "matrix" && v2.vartype = "matrix" then
        v1
    else if v1.vartype = "float" && v2.vartype="int" then
       v1
    else if v1.vartype = "int" && v2.vartype="float" then
       v2
    else if v1.vartype = "boolean" || v2.vartype = "boolean" then
     raise (Failure ("- operator cannot be applied to boolean operands"
                ^ ", but here used with '" ^ v1.varname ^ "', type: "
                ^ v1.vartype ^ " and '" ^ v2.varname ^ "', type: "
                ^ v2.vartype ^ " in function: '" ^ fname ^ "'"))
    else if v1.vartype <> v2.vartype then
     raise (Failure ("Type mismatch in binary operation between: '" ^
                v1.varname ^ "', type: " ^ v1.vartype ^ " and '" ^ v2.varname
                ^ "', type: " ^ v2.vartype ^ " in function: '" ^ fname
                ^ "'"))
    else v1
| Equal | Neq ->
 let v2 = string_of_expr globals locals fname e2 in
    if v1.vartype <> v2.vartype then
     raise (Failure ("Type mismatch in binary operation between: '" ^
                v1.varname ^ "', type: " ^ v1.vartype ^ " and '" ^ v2.varname
                ^ "', type: " ^ v2.vartype ^ " in function: '" ^ fname
                ^ "'"))
    else { varname = "null"; vartype = "boolean" }
  | Mul ->
 let v2 = string_of_expr globals locals fname e2 in
    if v1.vartype = "string" || v2.vartype = "string" then
     raise (Failure ("Operators '*', '/' " ^
                "cannot be applied to a string, in function: '" ^ fname ^ "'"))
  else if v1.vartype = "matrix" && v2.vartype = "int" then
        v1
  else if v1.vartype = "matrix" && v2.vartype = "float" then
        v1
  else if v1.vartype = "int" && v2.vartype = "matrix" then
        v2
```

```
    else if v1.vartype = "float" && v2.vartype = "matrix" then
          v2
    else if v1.vartype = "matrix" && v2.vartype = "matrix" then
          v1
    else if v1.vartype = "float" && v2.vartype="int" then
        v1
    else if v1.vartype = "int" && v2.vartype="float" then
        v2
    else if v1.vartype = "boolean" || v2.vartype = "boolean" then
     raise (Failure ("* operators cannot be applied to boolean operands"
                  ^ ", but here used with '" ^ v1.varname ^ "', type: "
                  ^ v1.vartype ^ " and '" ^ v2.varname ^ "', type: "
                  ^ v2.vartype ^ " in function: '" ^ fname ^ "'"))
    else if v1.vartype <> v2.vartype then
     raise (Failure ("Type mismatch in binary operation between: '" ^
                   v1.varname ^ "', type: " ^ v1.vartype ^ " and '" ^ v2.varname
                   ^ "', type: " ^ v2.vartype ^ " in function: '" ^ fname
                   ^ "'"))
    else v1
| Div ->
              let v2 = string_of_expr globals locals fname e2 in
                if v1.vartype = "string" || v2.vartype = "string" then
                  raise (Failure ("Operators '*', '/' " ^
                                "cannot be applied to a string, in function: '" ^ fname ^ "'"))
                else if v1.vartype = "matrix" && v2.vartype = "int" then
                      v1
                else if v1.vartype = "matrix" && v2.vartype = "float" then
                      v1
                else if v1.vartype = "int" && v2.vartype = "matrix" then
                      v2
                else if v1.vartype = "float" && v2.vartype = "matrix" then
                              v2
              else if v1.vartype = "float" && v2.vartype="int" then
                  v1
              else if v1.vartype = "int" && v2.vartype="float" then
                  v2

                else if v1.vartype = "matrix" && v2.vartype = "matrix" then
                     raise(Failure("/ operator cannot be applied to matrix operands"
                      ^ ", but here used with '" ^ v1.varname ^ "', type: "
                               ^ v1.vartype ^ " and '" ^ v2.varname ^ "', type: "
```

```
                    ^ v2.vartype ^ " in function: "' ^ fname ^ "'"))
                else if v1.vartype = "boolean" || v2.vartype = "boolean" then
                 raise (Failure ("/ operator cannot be applied to boolean operands"
                        ^ ", but here used with "' ^ v1.varname ^ "', type: "
                        ^ v1.vartype ^ " and "' ^ v2.varname ^ "', type: "
                        ^ v2.vartype ^ " in function: "' ^ fname ^ "'"))
                else if v1.vartype <> v2.vartype then
                 raise (Failure ("Type mismatch in binary operation between: "' ^
                        v1.varname ^ "', type: " ^ v1.vartype ^ " and "' ^
v2.varname
                        ^ "', type: " ^ v2.vartype ^ " in function: "' ^ fname
                        ^ "'"))
               else if v1.vartype ="matrix" && v2.vartype="matrix" then
                raise (Failure ("/ operator cannot be applied to matrix operands"
                        ^ ", but here used with "' ^ v1.varname ^ "', type: "
                        ^ v1.vartype ^ " and "' ^ v2.varname ^ "', type: "
                        ^ v2.vartype ^ " in function: "' ^ fname ^ "'"))

            else v1
    | Emul | Erdiv | Ldiv | Eldiv->
            let v2 = string_of_expr globals locals fname e2 in
      if v1.vartype = "matrix" && v2.vartype = "matrix" then
        v1
      else
       raise (Failure ("The operands for operators '.*', './', and '\\' " ^
              "must be matrices, in function: "' ^ fname ^ "'"))
     | Less | Leq | Greater | Geq ->
    let v2 = string_of_expr globals locals fname e2 in
      if v1.vartype = "string" || v2.vartype = "string" then
       raise (Failure ("Operators '<', '<=', '>', '>=' " ^
              "cannot be applied to a string, in function: "' ^ fname ^ "'"))
      else if v1.vartype = "boolean" || v2.vartype = "boolean" then
       raise (Failure ("<,<=,>,>= operators cannot be applied to boolean operands"
              ^ ", but here used with "' ^ v1.varname ^ "', type: "
              ^ v1.vartype ^ " and "' ^ v2.varname ^ "', type: "
              ^ v2.vartype ^ " in function: "' ^ fname ^ "'"))
            else if v1.vartype = "matrix" || v2.vartype = "matrix" then
       raise (Failure ("<,<=,>,>= operators cannot be applied to matrix operands"
              ^ ", but here used with "' ^ v1.varname ^ "', type: "
              ^ v1.vartype ^ " and "' ^ v2.varname ^ "', type: "
              ^ v2.vartype ^ " in function: "' ^ fname ^ "'"))
```

```
            else if v1.vartype <> v2.vartype then
              raise (Failure ("Type mismatch in binary operation between: "' ^
                              v1.varname ^ "', type: " ^ v1.vartype ^ " and "' ^ v2.varname
                              ^ "', type: " ^ v2.vartype ^ " in function: "' ^ fname
                              ^ """))
            else { varname = "null"; vartype = "boolean" }
           | And | Or ->
          let v2 = string_of_expr globals locals fname e2 in
            if v1.vartype = "boolean" && v2.vartype = "boolean" then
               v1
            else
               raise (Failure ("and, or operators can only be applied to boolean
expressions"
                               ^ ", but here used with "' ^ v1.varname ^ "', type: "
                               ^ v1.vartype ^ " and "' ^ v2.varname ^ "', type: "
                               ^ v2.vartype ^ " in function: "' ^ fname ^ """))
         | Power ->
          let v2 = string_of_expr globals locals fname e2 in
          if v1.vartype = "int" && v2.vartype = "int" then
             v1
          else if v1.vartype = "float" && v2.vartype = "float" then
             v1
          else if v1.vartype = "matrix" && v2.vartype = "int" then
             v1
          else if v1.vartype = "matrix" && v2.vartype = "float" then
             v1
            else if v1.vartype = "float" && v2.vartype="int" then
               v1
            else if v1.vartype = "int" && v2.vartype="float" then
               v2
          else
                 raise (Failure ("For ^ operator, if the first operand is int or float, the type of
the second argument must be the same as the first.\n"^
                 "If the first operand is matrix, the type of the second argument must be
either int or float.\n"
                 ^ "but here got "' ^ v1.vartype ^ ": " ^ v1.varname ^ "' and "'
                 ^ v2.vartype ^ ": " ^ v2.varname ^ "' in function: "' ^ fname ^ """)))
      | Assign(v, e) ->
          if NameMap.mem v locals then
             let var1 = NameMap.find v locals in
             let var2 = string_of_expr globals locals fname e in
```

```
            if var1.vartype <> var2.vartype then
               raise (Failure ("Type mismatch in assignment operation between: "'
                  ^ var1.varname ^ '", type: " ^ var1.vartype ^ " and "'
                  ^ var2.varname ^ '", type: " ^ var2.vartype
                  ^ " in function: "' ^ fname ^ '"'))
            else var1
         else if NameMap.mem v globals then
            let var1 = NameMap.find v globals in
          let var2 = string_of_expr globals locals fname e in
            if var1.vartype <> var2.vartype then
               raise (Failure ("Type mismatch in assignment operation between: "'
                  ^ var1.varname ^ '", type: " ^ var1.vartype ^ " and "'
                  ^ var2.varname ^ '", type: " ^ var2.vartype
                  ^ " in function: "' ^ fname ^ '"'))
            else var1
         else raise (Failure ("Undeclared identifier found in assignment: "'
            ^ v ^ '" in function: "' ^ fname ^ '"'))
    | AssignMatrix(id, indexList, value) ->
          let varvalue = string_of_expr globals locals fname value in
          if (List.length indexList !=2) then raise (Failure("Matrix assignment for " ^id ^"
requires 2 indices"))
          else
          let checkIndices globals locals fname matrix indexList =
                        List.iter
                        (fun exp ->
                                let v = string_of_expr globals locals fname exp in
                                if v.vartype <> "int" then
                                        raise (Failure ("Invalid index value: all indices "
                                   ^ " must be int type. matrix: "' ^ matrix ^ '" ;"
                                     ^"function: "' ^ fname ^ '"'))) indexList
            in

          let chkIn=checkIndices globals locals fname id indexList in
          blackhole chkIn;
          if  varvalue.vartype <> "float" && varvalue.vartype <> "int" then
             raise (Failure ("Type mismatch in matrix assignment."
                ^ " The value must be of type float or int but here used with type "'
                ^ varvalue.vartype ^ '" in function: "' ^ fname ^ '"'))
          else
             if NameMap.mem id locals then
                let varId = NameMap.find id locals in
```

```
                if varId.vartype <> "matrix" then
                   raise (Failure ("Expecting matrix type but got "' ^ varId.vartype ^
                      '" for variable: "' ^ varId.varname ^ " in function: "' ^ fname ^ '"'))
                else { varname = "null";vartype = "int" }
             else if NameMap.mem id globals then
                let varId = NameMap.find id globals in
                if varId.vartype <> "matrix" then
                raise (Failure ("Expecting matrix type but got "' ^ varId.vartype ^
                   '" for variable: "' ^ varId.varname ^ " in function: "' ^ fname ^ '"'))
                else { varname = "null";vartype = "int" }
                 else raise (Failure ("Undeclared identifier found in assignment: "'
                     ^ id ^ '" in function: "' ^ fname ^ '"'))
    | MatrixAccess(id, indexList) ->
        if (List.length indexList !=2) then raise (Failure("Matrix access for " ^id ^" requires 2
indices"))
           else
        let checkIndices globals locals fname matrix indexList =
                List.iter
                      (fun exp ->
                      let v = string_of_expr globals locals fname exp in
                         if v.vartype <> "int" then
                         raise (Failure ("Invalid index value: all indices "
                         ^ " must be int type. matrix: "' ^ matrix ^ '" ;"
                        ^"function: "' ^ fname ^ '"'))) indexList
          in

         let chkIn=checkIndices globals locals fname id indexList in
         blackhole chkIn;
         if NameMap.mem id locals then
            let varId = NameMap.find id locals in
            if varId.vartype <> "matrix" then
               raise (Failure ("Expecting matrix type but got "' ^ varId.vartype ^
                  '" for variable: "' ^ varId.varname ^ " in function: "' ^ fname ^ '"'))
            else { varname = id;vartype = "float" }
        else if NameMap.mem id globals then
            let varId = NameMap.find id globals in
            if varId.vartype <> "matrix" then
               raise (Failure ("Expecting matrix type but got "' ^ varId.vartype ^
                  '" for variable: "' ^ varId.varname ^ " in function: "' ^ fname ^ '"'))
            else { varname = id;vartype = "float" }
```

```
          else raise (Failure ("Undeclared identifier:" ^id^" found in function: "' ^ fname ^
"'"))
      | MatrixRAccess(id, i1, i2, j1, j2) ->
                let ti1=string_of_expr globals locals fname i1 in
                let ti2=string_of_expr globals locals fname i2 in
                let tj1=string_of_expr globals locals fname j1 in
                let tj2=string_of_expr globals locals fname j2 in

               if    ti1.vartype<>"int"    ||    ti2.vartype<>"int"    ||    tj1.vartype<>"int"    ||
tj2.vartype<>"int" then raise (Failure("The range indices in " ^id^ " must by type int."))
              else
              if NameMap.mem id locals then
                   let varId = NameMap.find id locals in
                   if varId.vartype <> "matrix" then
                          raise (Failure ("Expecting matrix type but got "' ^ varId.vartype ^
                             "' for variable: "' ^ varId.varname ^ " in function: "' ^ fname ^ "'"))
                     else { varname = id;vartype = "matrix" }
                  else if NameMap.mem id globals then
                  let varId = NameMap.find id globals in
                  if varId.vartype <> "matrix" then
                          raise (Failure ("Expecting matrix type but got "' ^ varId.vartype ^
                             "' for variable: "' ^ varId.varname ^ " in function: "' ^ fname ^ "'"))
                       else { varname = id;vartype = "matrix" }
          else raise (Failure ("Undeclared identifier:"^id^" found in function: "' ^ fname ^
"'"))
    | Newmat(row, col) ->
                let varRow = string_of_expr globals locals fname row in
                let varCol = string_of_expr globals locals fname col in
               if varRow.vartype <> varCol.vartype || varRow.vartype <> "int" then
                  raise (Failure ("Type mismatch in creating new matrix. Both row and
column"
                     ^ " sizes must be of type int; in function: "' ^ fname ^ "'"))
                else
                  { varname = "newmat";vartype = "matrix" }
     | Call("mod", actuals) ->
          let actuals = List.fold_left
             (fun actuals actual ->
                  let v = string_of_expr globals locals fdecl.fname actual in
                     v :: actuals) [] actuals
             in
             if (List.length actuals) <> 2 then
```

```
            raise (Failure ("'mod' function takes 2 parameters."
               ^ ", called from function: "' ^ fname ^ "'"))
         else
            let param1 = List.nth actuals 1 in
            let param2 = List.nth actuals 0 in
               if param1.vartype = "int" && param2.vartype = "int" then
                  { varname = "mod" ; vartype = "int" }
               else if param1.vartype = "float" && param2.vartype = "float" then
                  { varname = "mod" ; vartype = "float" }
               else
                  raise (Failure ("Arguments passed to function 'mod' must be all of"
                  ^ " either int or float type,"
                  ^ " but here got "' ^ param1.vartype ^ "' and "'
                        ^ param2.vartype ^ "' called from function: "' ^ fname ^ "'"))
| Call("disp", actuals) ->
    let actuals = List.fold_left
       (fun actuals actual ->
             let v = string_of_expr globals locals fdecl.fname actual in
                v :: actuals) [] actuals
       in
       if (List.length actuals) <>1 then
          raise (Failure ("Wrong number of arguments passed to function 'disp"'
             ^ ", called from function: "' ^ fname ^ "'"))


       else { varname = "disp" ; vartype = "void" }

| Call("setFormat", actuals) ->
    let actuals = List.fold_left
       (fun actuals actual ->
             let v = string_of_expr globals locals fdecl.fname actual in
                v :: actuals) [] actuals
       in
       if (List.length actuals) <> 1 then
          raise (Failure ("Wrong number of arguments passed to function 'setFormat"'
             ^ ", called from function: "' ^ fname ^ "'"))

       else
       let param= List.hd actuals in
       if param.vartype<>"string" then
```

```
                raise(Failure("The setFormat function only takes the following string parameters: 'short', 'shortE','long', 'longE'"))
            else if param.varname <>"short" && param.varname<>"shortE" && param.varname<>"long" && param.varname<>"longE" then
                raise(Failure("The setFormat function only takes the following string parameters: 'short', 'shortE','long', 'longE'"))
            else { varname = "setFormat" ; vartype = "void" }
    | Call("tostring", actuals) ->
        let actuals = List.fold_left
            (fun actuals actual ->
                let v = string_of_expr globals locals fdecl.fname actual in
                    v :: actuals) [] actuals
            in
            if (List.length actuals) <> 1 then
                raise (Failure ("'tostring' function only takes one parameter."
                    ^ ", called from function: "' ^ fname ^ "'"))
            else
                let param = List.hd actuals in
                    if param.vartype <> "matrix" then
                        { varname = "tostring" ; vartype = "string" }
                    else
                        raise (Failure ("Argument passed to function 'tostring' cannot be "
                            ^ "matrix type, called from function: "' ^ fname ^ "'"))
    | Call("load", actuals) ->
            let actuals = List.fold_left
                (fun actuals actual ->
                let v = string_of_expr globals locals fdecl.fname actual in
                    v :: actuals) [] actuals
                in
                if (List.length actuals) <> 1 then
                    raise (Failure ("function 'load' only takes one string parameter "
                        ^ ", called from function: "' ^ fname ^ "'"))
                else
                        let param= List.hd actuals in
                if param.vartype<>"string" then
                        raise(Failure("The 'load' function only takes the one parameter of type string"))
                else
                { varname = "load" ; vartype = "matrix" }
    | Call("save", actuals) ->
                        let actuals = List.fold_left
```

```
                    (fun actuals actual ->
                       let v = string_of_expr globals locals fdecl.fname actual in
                          v :: actuals) [] actuals
                    in
                    if (List.length actuals) <> 2 then
                       raise (Failure ("Function 'save takes 2 argument"
                          ^ ", called from function: "' ^ fname ^ '"'))
                    else
                       (*actuals list in reverse order*)
                       let param2= (List.nth actuals 0) in
                       let param1= (List.nth actuals 1) in
                       if param1.vartype <>"matrix" then
                              raise(Failure(param1.vartype^"The first argument for the
'save' funnction must be a matrix"))
                       else
                              if param2.vartype<>"string" then
                                     raise(Failure("The second argument for the 'save'
function must be a string"))
                              else
                                     { varname = "save" ; vartype = "void" }

    | Call("width", actuals) ->
       let actuals = List.fold_left
          (fun actuals actual ->
                let v = string_of_expr globals locals fdecl.fname actual in
                   v :: actuals) [] actuals
          in
          if (List.length actuals) <> 1 then
             raise (Failure ("'width' function takes only one parameter."
                ^ ", called from function: "' ^ fname ^ '"'))
          else
             let param = List.hd actuals in
                if param.vartype = "matrix" then
                   { varname = "width" ; vartype = "int" }
                else
                   raise (Failure ("Argument passed to function 'width' must be of "
                   ^ "type matrix, called from function: "' ^ fname ^ '"'))

    | Call("inv", actuals) ->
       let actuals = List.fold_left
          (fun actuals actual ->
```

```
            let v = string_of_expr globals locals fdecl.fname actual in
              v :: actuals) [] actuals
        in
        if (List.length actuals) <> 1 then
          raise (Failure (""inv' function takes only one parameter. "
            ^ ", called from function: "' ^ fname ^ '"'))
        else
          let param = List.hd actuals in
            if param.vartype = "matrix" then
              { varname = "inv" ; vartype = "matrix" }
            else
              raise (Failure ("Argument passed to function 'inv' must be of "
              ^ "type matrix, called from function: "' ^ fname ^ '"'))
  | Call("height", actuals) ->
    let actuals = List.fold_left
        (fun actuals actual ->
            let v = string_of_expr globals locals fdecl.fname actual in
              v :: actuals) [] actuals
        in
        if (List.length actuals) <> 1 then
          raise (Failure (""height' function takes only one parameter"
            ^ ", called from function: "' ^ fname ^ '"'))
        else
          let param = List.hd actuals in
            if param.vartype = "matrix" then
              { varname = "height" ; vartype = "int" }
            else
              raise (Failure ("Argument passed to function 'height' must be of  "
              ^ "type matrix, called from function: "' ^ fname ^ '"'))
  | Call("sqrt", actuals) ->
    let actuals = List.fold_left
        (fun actuals actual ->
            let v = string_of_expr globals locals fdecl.fname actual in
              v :: actuals) [] actuals
        in
        if (List.length actuals) <> 1 then
          raise (Failure ("function 'sqrt' takes just one parameter"
            ^ ", called from function: "' ^ fname ^ '"'))
        else
          let param = List.hd actuals in
            if param.vartype = "float" || param.vartype = "int" then
```

```
                { varname = "sqrt" ; vartype = "float" }
              else
                raise (Failure ("Argument passed to function 'sqrt' can only be "
                  ^ "either float or int type, called from function: "' ^ fname ^ """))
| Call("abs", actuals) ->
        let actuals = List.fold_left
          (fun actuals actual ->
            let v = string_of_expr globals locals fdecl.fname actual in
              v :: actuals) [] actuals
          in
          if (List.length actuals) <> 1 then
            raise (Failure ("function 'abs' takes just one parameter"
              ^ ", called from function: "' ^ fname ^ """))
          else
            let param = List.hd actuals in
              if param.vartype = "float" || param.vartype = "int" then
                { varname = "abs" ; vartype = "float" }
              else
                raise (Failure ("Argument passed to function 'abs' can only be "
                ^ "either float or int type, called from function: "' ^ fname ^ """))
| Call("exp", actuals) ->
        let actuals = List.fold_left
          (fun actuals actual ->
            let v = string_of_expr globals locals fdecl.fname actual in
              v :: actuals) [] actuals
          in
          if (List.length actuals) <> 1 then
            raise (Failure ("function 'exp' takes just one parameter"
              ^ ", called from function: "' ^ fname ^ """))
          else
            let param = List.hd actuals in
              if param.vartype = "float" || param.vartype = "int" then
                { varname = "exp" ; vartype = "float" }
              else
                raise (Failure ("Argument passed to function 'exp' can only be "
                ^ "either float or int type, called from function: "' ^ fname ^ """))
| Call("log", actuals) ->
        let actuals = List.fold_left
          (fun actuals actual ->
            let v = string_of_expr globals locals fdecl.fname actual in
              v :: actuals) [] actuals
```

```
            in
            if (List.length actuals) <> 1 then
               raise (Failure ("function 'log' takes just one parameter"
                  ^ ", called from function: "' ^ fname ^ "'"))
            else
               let param = List.hd actuals in
                  if param.vartype = "float" || param.vartype = "int" then
                     { varname = "log" ; vartype = "float" }
                  else
                     raise (Failure ("Argument passed to function 'log' can only be "
                      ^ "either float or int type, called from function: "' ^ fname ^ "'"))
| Call("log10", actuals) ->
         let actuals = List.fold_left
            (fun actuals actual ->
             let v = string_of_expr globals locals fdecl.fname actual in
                v :: actuals) [] actuals
            in
            if (List.length actuals) <> 1 then
               raise (Failure ("function 'log' takes just one parameter"
                  ^ ", called from function: "' ^ fname ^ "'"))
            else
               let param = List.hd actuals in
                  if param.vartype = "float" || param.vartype = "int" then
                     { varname = "log" ; vartype = "float" }
                  else
                     raise (Failure ("Argument passed to function 'log' can only be "
                      ^ "either float or int type, called from function: "' ^ fname ^ "'"))
| Call("sin", actuals) ->
         let actuals = List.fold_left
            (fun actuals actual ->
             let v = string_of_expr globals locals fdecl.fname actual in
                v :: actuals) [] actuals
            in
            if (List.length actuals) <> 1 then
               raise (Failure ("function 'sin' takes just one parameter"
                  ^ ", called from function: "' ^ fname ^ "'"))
            else
               let param = List.hd actuals in
                  if param.vartype = "float" || param.vartype = "int" then
                     { varname = "sin" ; vartype = "float" }
                  else
```

```
                raise (Failure ("Argument passed to function 'sin' can only be "
                ^ "either float or int type, called from function: "' ^ fname ^ '"'))
| Call("cos", actuals) ->
        let actuals = List.fold_left
            (fun actuals actual ->
             let v = string_of_expr globals locals fdecl.fname actual in
                v :: actuals) [] actuals
            in
            if (List.length actuals) <> 1 then
               raise (Failure ("function 'cos' takes just one parameter"
                  ^ ", called from function: "' ^ fname ^ '"'))
            else
               let param = List.hd actuals in
                  if param.vartype = "float" || param.vartype = "int" then
                     { varname = "cos" ; vartype = "float" }
                  else
                     raise (Failure ("Argument passed to function 'cos' can only be "
                     ^ "either float or int type, called from function: "' ^ fname ^ '"'))
| Call("tan", actuals) ->
        let actuals = List.fold_left
            (fun actuals actual ->
             let v = string_of_expr globals locals fdecl.fname actual in
                v :: actuals) [] actuals
            in
            if (List.length actuals) <> 1 then
               raise (Failure ("function 'tan' takes just one parameter"
                  ^ ", called from function: "' ^ fname ^ '"'))
            else
               let param = List.hd actuals in
                  if param.vartype = "float" || param.vartype = "int" then
                     { varname = "tan" ; vartype = "float" }
                  else
                     raise (Failure ("Argument passed to function 'tan' can only be "
                     ^ "either float or int type, called from function: "' ^ fname ^ '"'))
| Call("asin", actuals) ->
        let actuals = List.fold_left
            (fun actuals actual ->
             let v = string_of_expr globals locals fdecl.fname actual in
                v :: actuals) [] actuals
            in
            if (List.length actuals) <> 1 then
```

```
                raise (Failure ("function 'sqrt' takes just one parameter"
                    ^ ", called from function: "' ^ fname ^ "'"))
            else
                let param = List.hd actuals in
                    if param.vartype = "float" || param.vartype = "int" then
                        { varname = "asin" ; vartype = "float" }
                    else
                        raise (Failure ("Argument passed to function 'asin' can only be "
                        ^ "either float or int type, called from function: "' ^ fname ^ "'"))
| Call("acos", actuals) ->
        let actuals = List.fold_left
            (fun actuals actual ->
             let v = string_of_expr globals locals fdecl.fname actual in
                v :: actuals) [] actuals
            in
            if (List.length actuals) <> 1 then
                raise (Failure ("function 'acos' takes just one parameter"
                    ^ ", called from function: "' ^ fname ^ "'"))
            else
                let param = List.hd actuals in
                    if param.vartype = "float" || param.vartype = "int" then
                        { varname = "acos" ; vartype = "float" }
                    else
                        raise (Failure ("Argument passed to function 'acos' can only be "
                        ^ "either float or int type, called from function: "' ^ fname ^ "'"))
| Call("atan", actuals) ->
            let actuals = List.fold_left
                (fun actuals actual ->
                        let v = string_of_expr globals locals fdecl.fname actual in
                            v :: actuals) [] actuals
                in
                if (List.length actuals) <> 1 then
                    raise (Failure ("function 'atan' takes just one parameter"
                        ^ ", called from function: "' ^ fname ^ "'"))
                else
                    let param = List.hd actuals in
                        if param.vartype = "float" || param.vartype = "int" then
                            { varname = "atan" ; vartype = "float" }
                        else
                            raise (Failure ("Argument passed to function 'atan' can only be "
                    ^ "either float or int type, called from function: "' ^ fname ^ "'"))
```

```
| Call("atan2", actuals) ->
            let actuals = List.fold_left
               (fun actuals actual ->
                     let v = string_of_expr globals locals fdecl.fname actual in
                        v :: actuals) [] actuals
               in
               if (List.length actuals) <> 2 then
                  raise (Failure ("function 'atan2' takes 2 parameter"
                     ^ ", called from function: "' ^ fname ^ "'"))
               else
                  let param = List.hd actuals in
                     if param.vartype = "float" || param.vartype = "int" then
                        { varname = "atan" ; vartype = "float" }
                     else
                        raise (Failure ("Argument passed to function 'atan' can only be "
                  ^ "either float or int type, called from function: "' ^ fname ^ "'"))
| Call(f, actuals) ->
   let fdecl =
      if f = "main" then
         raise (Failure ("main function cannot be called by " ^
                              "function: "' ^ fname ^ "'"))
      else
            try NameMap.find f func_decls
               with Not_found -> raise (Failure ("Undefined function: "'
                  ^ f ^ "' in function call: "' ^ fname ^ "'"))
            in
            if f <> fname then
            let actuals = List.fold_left
               (fun actuals actual ->
                     let v = string_of_expr globals locals fdecl.fname actual in
                        v :: actuals) [] actuals
               in
               string_of_fdecl globals fdecl (List.rev actuals);
               { varname = fdecl.retname ; vartype = fdecl.rettype }
      else
         let actuals = List.fold_left
               (fun actuals actual ->
                     let v = string_of_expr globals locals fdecl.fname actual in
                        v :: actuals) [] actuals
               in
      (try List.iter2 (fun formal actual ->
```

```
            if formal.vartype <> actual.vartype then
              raise (Failure ("Type mismatch in argument passing between: "'
              ^ formal.varname ^ "', type: " ^ formal.vartype ^ " and "'
              ^ actual.varname ^ "', type: " ^ actual.vartype
              ^ " in function: "' ^ fdecl.fname ^ """)))
             fdecl.formals (List.rev actuals)
        (* Check invalid number of arguments *)
        with Invalid_argument(_) ->
          raise (Failure ("Wrong number of arguments passed to function: "'
            ^ fdecl.fname ^ "' from function: "' ^ fname ^ """)));
        { varname = fdecl.retname ; vartype = fdecl.rettype }
in

let check_for_expr v l globals locals fname =
  if v.vartype <> "int" or v.vartype <> "float" then
    let varList = List.map (string_of_expr globals locals fname) l
        in
      List.iter (fun var -> if var.vartype <> v.vartype then
          raise (Failure ("For loop type mismatch between "' ^ v.varname
          ^ "', type: " ^ v.vartype ^ " and "' ^ var.varname
          ^ "', type: " ^ var.vartype ^ " in function: "' ^ fname ^ """))) varList;
      else
        raise (Failure ("Only int/float is allowed in the for loop. Id "'
          ^ v.varname ^ "', type: " ^ v.vartype ^ " found in function: "' ^ fname ^ """))
in

let rec string_of_for_expr globals locals fname = function
  Assigna(v, e1, e2) ->
  if NameMap.mem v locals then
    let var = NameMap.find v locals in
    check_for_expr var [e1;e2;] globals locals fname
  else if NameMap.mem v globals then
        let var = NameMap.find v globals in
        check_for_expr var [e1;e2;] globals locals fname
      else raise (Failure ("Undeclared identifier in for loop: "' ^ v
        ^ "' in function: "' ^ fname ^ """))
|  Assignb(v, e1, e2, e3) ->
  if NameMap.mem v locals then
    let var = NameMap.find v locals in
    check_for_expr var [e1;e2;e3;] globals locals fname
      else if NameMap.mem v globals then
```

```
            let var = NameMap.find v globals in
            check_for_expr var [e1;e2;e3] globals locals fname
    else raise (Failure ("Undeclared identifier in for loop: "' ^ v
      ^ "' in function: "' ^ fname ^ """))
in
let checkDimension matrix fname l =
    let rec loop l =
      if (List.length l) = 1 then
        List.length (List.hd l)
      else
        let c1 = List.length (List.hd l)
        and c2 = loop (List.tl l) in
        if (c1 <> c2) then
          raise (Failure ("Invalid matrix initialization: number of columns"
          ^ " in all rows must be same. matrix: "' ^ matrix ^ "' ;"
          ^"function: "' ^ fname ^ """))
        else c1
      in loop l
in
let checkMatrixValues globals locals matrix fname l =
    let flatL = List.concat l in
      List.iter
      (fun exp ->
      let v = string_of_expr globals locals fname exp in
        if v.vartype <> "float" && v.vartype<> "int" then
          raise (Failure ("Invalid matrix value: all values passed to "
            ^ " matrix must be float type. matrix: "' ^ matrix ^ "' ;"
          ^"function: "' ^ fname ^ """))) flatL
in
let rec string_of_stmt globals locals fname = function
    Block(stmts) ->
      List.iter (string_of_stmt globals locals fname) stmts
  | Expr(expr) -> let e = string_of_expr globals locals fname expr in blackhole e
  | If(e, s1, s2) ->  let v = string_of_expr globals locals fname e in
      if v.vartype <> "boolean" then
        raise (Failure ("if expression must be evaluated to boolean, "
          ^ "but here found with "' ^ v.vartype ^ "' ;"
          ^ "function: "' ^ fname ^ """))
      else
        string_of_stmt globals locals fname s1;
        string_of_stmt globals locals fname s2;
```

```
| Ifelseif(e, s1, eif_list, s2) ->
    let v = string_of_expr globals locals fname e in
    if v.vartype <> "boolean" then
       raise (Failure ("if expression must be evaluated to boolean, "
          ^ "but here found with "' ^ v.vartype ^ "' ;"
          ^ "function: "' ^ fname ^ """))
    else
       let rec string_of_elseif eif =
          let en = string_of_expr globals locals fname eif.elseif_expr in
             if en.vartype <> "boolean" then
                raise (Failure ("if expression must be evaluated to boolean, "
                   ^ "but here found with "' ^ v.vartype ^ "' ;"
                   ^ "function: "' ^ fname ^ """))
             else
                string_of_stmt globals locals fname eif.elseif_stmt;
       in
          string_of_stmt globals locals fname s1;
          List.iter string_of_elseif eif_list;
          string_of_stmt globals locals fname s2;
| For(e, s) ->loop_flag :=  true;
       string_of_for_expr globals locals fname e;
       string_of_stmt globals locals fname s;
       loop_flag := false;
| While(e, s) -> loop_flag:=true;
    let v = string_of_expr globals locals fname e
    in
    if v.vartype <> "boolean" then
       raise (Failure ("while expression must be evaluated to boolean, "
          ^ "but here found with "' ^ v.vartype ^ "' ;"
          ^ "function: "' ^ fname ^ """))
    else
       string_of_stmt globals locals fname s;
    loop_flag:=false;
| MatrixInit(id, valuesList) ->
    if NameMap.mem id locals then
       let varId = NameMap.find id locals in
       if varId.vartype <> "matrix" then
             raise (Failure ("Expecting matrix type but got "' ^ varId.vartype ^
                "' for variable: "' ^ varId.varname ^ " in function: "' ^ fname ^ """))
          else
             let col = checkDimension id fname valuesList in
```

```
                blackhole col;
                checkMatrixValues globals locals id fname valuesList
          else if NameMap.mem id globals then
          let varId = NameMap.find id globals in
          if varId.vartype <> "matrix" then
               raise (Failure ("Expecting matrix type but got "' ^ varId.vartype ^
                 '" for variable: "' ^ varId.varname ^ " in function: "' ^ fname ^ """))
            else
               let col = checkDimension id fname valuesList in
                 blackhole col;
                 checkMatrixValues globals locals id fname valuesList
          else raise (Failure ("Undeclared identifier:"^id^" found in function: "' ^ fname ^
"""))
     | Break -> if !loop_flag = false then
                      raise (Failure ("Misplaced break statement is found outside of a
loop"))
                else
                ()

     | Continue ->if !loop_flag = false then
                      raise (Failure ("Misplaced continue statement is found outside of a
loop"))
                else
                ()
  in
  let locals =
     try List.fold_left2
        (fun locals formal actual ->
          if formal.vartype = actual.vartype then
            if NameMap.mem formal.varname locals then
              raise (Failure ("Formal variable: "' ^ formal.varname ^
                '" has already been defined in the function: "' ^
                fdecl.fname ^ '" parameter"))
            else NameMap.add formal.varname formal locals
          else raise (Failure ("Type mismatch in argument passing between: "'
                ^ formal.varname ^ '", type: " ^ formal.vartype ^ " and "'
                ^ actual.varname ^ '", type: " ^ actual.vartype
                ^ " in function: "' ^ fdecl.fname ^ """)))
            NameMap.empty fdecl.formals actuals
        (* Check invalid number of arguments *)
     with Invalid_argument(_) ->
```

```
            raise (Failure ("Wrong number of arguments passed to function: "
              ^ fdecl.fname ^ """))
    in
    let locals = List.fold_left
            (fun locals local ->
        if NameMap.mem local.varname locals then
            raise (Failure ("Local variable: "" ^ local.varname ^
              "" has already been defined in the function: "" ^
              fdecl.fname ^ """))
        else NameMap.add local.varname local locals) locals fdecl.locals
      in
      if fdecl.retname <> "" then
        if NameMap.mem fdecl.retname locals then
          raise (Failure ("Return variable: "" ^ fdecl.retname ^
            "" has already been defined in the function: "" ^
            fdecl.fname ^ """))
        else
          let ret = { varname = fdecl.retname; vartype = fdecl.rettype } in
          let locals = NameMap.add ret.varname ret locals in
          List.iter (string_of_stmt globals locals fdecl.fname) fdecl.body
      else
        List.iter (string_of_stmt globals locals fdecl.fname) fdecl.body
  in
  let globals = List.fold_left
    (fun globals vdecl ->
      if NameMap.mem vdecl.varname globals then
        raise (Failure ("Global variable : "" ^ vdecl.varname ^
              "" has already been defined."))
      else
        NameMap.add vdecl.varname vdecl globals) NameMap.empty vars
  in
    try
      string_of_fdecl globals (NameMap.find "main" func_decls) []
    with Not_found ->
      raise (Failure ("There is no main() function"))
```

Javawriter.ml

```
open Ast

let import_decl = "
import java.io.File;
```

```
import java.util.Formatter;
import java.util.Scanner;
import Jama.*;
"
let class_decl = "\n"^
"enum OPERATION {ADD, SUB, MUL, DIV, EMUL, ELDIV, ERDIV, LDIV, POW};\n"^
"static String numFormat = \"short\";\n"^
"    static void setNumFormat(String s){\n"^
"        numFormat=s;\n"^
"    }\n"^
"    static void disp(Object o) {\n"^
"        if (o instanceof String) {\n"^
"            System.out.println((String) o);\n"^
"        } else if (o instanceof Integer) {\n"^
"            Integer a = (Integer) o;\n"^
"            System.out.println(a.toString());\n"^
"        } else if (o instanceof Double) {\n"^
"            if (numFormat.equals(\"short\")) {\n"^
"                String fString = \"%\" + \"01\" + \".\" + 4 + \"f\\n\";\n"^
"                Double a = (Double) o;\n"^
"               System.out.format(fString, a);\n"^
"            } else if (numFormat.equals(\"shortE\")) {\n"^
"                String fString = \"%\" + \"1\" + \".\" + 4 + \"E\";\n"^
"                Double a = (Double) o;\n"^
"                System.out.format(fString, a);\n"^
"            }else if (numFormat.equals(\"long\")){\n"^
"                String fString = \"%\" + \"1\" + \".\" + 10 + \"f\";\n"^
"                Double a = (Double) o;\n"^
"                System.out.format(fString, a);\n"^
"            }else if (numFormat.equals(\"longE\")){\n"^
"                String fString = \"%\" + \"01\" + \".\" + 10 + \"E\";\n"^
"                Double a = (Double) o;\n"^
"                System.out.format(fString, a);\n"^
"            }\n"^
"        }else if (o instanceof Matrix){\n"^
"            Matrix m=(Matrix) o;\n"^
"            String conversion=\"\";\n"^
"            int numDecimal=4;\n"^
"            if (numFormat.equals(\"short\") || numFormat.equals(\"shortE\")){\n"^
"                numDecimal=4;\n"^
"                if (numFormat.equals(\"short\")){\n"^
```

```
"                conversion=\"f\";\n"^
"             }else{\n"^
"                conversion=\"E\";\n"^
"             }\n"^
"          }else if(numFormat.equals(\"long\") || numFormat.equals(\"longE\")){\n"^
"             numDecimal=15;\n"^
"             if (numFormat.equals(\"long\")){\n"^
"                conversion=\"f\";\n"^
"             }else{\n"^
"                conversion=\"E\";\n"^
"             }\n"^
"          }\n"^
"          String formatStr=\"%\"+\"1\"+\".\"+numDecimal+conversion;\n"^
"          int maxWidth=0;\n"^
"          for (int i=0; i<m.getRowDimension(); i++){\n"^
"             for (int j=0; j<m.getColumnDimension(); j++){\n"^
"                String testStr=String.format(formatStr, m.get(i, j));\n"^
"               if (testStr.length()>maxWidth){\n"^
"                   maxWidth=testStr.length();\n"^
"                }\n"^
"             }\n"^
"          }\n"^
"          maxWidth+=6;\n"^
"          for (int i=0; i<m.getRowDimension(); i++){\n"^
"             for (int j=0; j<m.getColumnDimension(); j++){\n"^
"                formatStr=\"%\"+maxWidth+\".\"+numDecimal+conversion;\n"^
"                String testStr=String.format(formatStr, m.get(i, j));\n"^
"                System.out.print(testStr);\n"^
"             }\n"^
"             System.out.println();\n"^
"          }\n"^
"       }\n"^
"    }\n"^
"    static Matrix matrixOperations(Object op1, Object op2, OPERATION op) {\n"^
"         if (op1 instanceof Matrix && (op2 instanceof Integer || op2 instanceof Double))
{\n"^
"          Matrix temp1 = (Matrix) op1;\n"^
"                              Matrix temp2 = new Matrix(temp1.getRowDimension(),
temp1.getColumnDimension(), Double.parseDouble(op2.toString()));\n"^
"                                 Matrix temp3=new Matrix(temp1.getRowDimension(),
temp1.getColumnDimension());\n"^
```

```
"          switch (op) {\n"^
"             case ADD:\n"^
"                return temp1.plus(temp2);\n"^
"             case SUB:\n"^
"                return temp1.minus(temp2);\n"^
"             case MUL:\n"^
"                return temp1.times(Double.parseDouble(op2.toString()));\n"^
"             case DIV:\n"^
"                for (int i = 0; i < temp1.getRowDimension(); i++) {\n"^
"                   for (int j = 0; j < temp1.getColumnDimension(); j++) {\n"^
"                      double product = temp1.get(i, j) / temp2.get(i,j);\n"^
"                      temp3.set(i, j, product);\n"^
"                   }\n"^
"                }\n"^
"                return temp3;\n"^
"             case POW:\n"^
"                for (int i = 0; i < temp1.getRowDimension(); i++) {\n"^
"                   for (int j = 0; j < temp1.getColumnDimension(); j++) {\n"^
"                      double product = Math.pow(temp1.get(i, j), temp2.get(i,j));\n"^
"                      temp1.set(i, j, product);\n"^
"                   }\n"^
"                }\n"^
"                return temp1;\n"^
"          }\n"^
"           }else if((op1 instanceof Integer || op1 instanceof Double) && op2 instanceof
Matrix) {\n"^
"            Matrix temp1 = (Matrix) op2;\n"^
"                                  Matrix temp2 = new Matrix(temp1.getRowDimension(),
temp1.getColumnDimension(), Double.parseDouble(op1.toString()));\n"^
"                  Matrix              temp3=new              Matrix(temp1.getRowDimension(),
temp1.getColumnDimension());\n"^
"             switch (op) {\n"^
"              case ADD:\n"^
"                 return temp1.plus(temp2);\n"^
"              case SUB:\n"^
"                 return temp1.minus(temp2);\n"^
"              case MUL:\n"^
"                 return temp1.times(Double.parseDouble(op1.toString()));\n"^
"              case DIV:\n"^
"                 for (int i = 0; i < temp1.getRowDimension(); i++) {\n"^
"                    for (int j = 0; j < temp1.getColumnDimension(); j++) {\n"^
```

```
"                        double product = temp2.get(i, j) / temp1.get(i,j);\n"^
"                        temp3.set(i, j, product);\n"^
"                    }\n"^
"                }\n"^
"                return temp3;\n"^
"            }\n"^
"        }else if (op1 instanceof Matrix && op2 instanceof Matrix) {\n"^
"        Matrix temp1 = (Matrix) op1;\n"^
"        Matrix temp2 = (Matrix) op2;\n"^
"        switch (op) {\n"^
"            case ADD:\n"^
"                return temp1.plus(temp2);\n"^
"            case SUB:\n"^
"                return temp1.minus(temp2);\n"^
"            case MUL:\n"^
"                return temp1.times(temp2);\n"^
"            case EMUL:\n"^
"                return temp1.arrayTimes(temp2);\n"^
"            case ELDIV:\n"^
"                return temp1.arrayLeftDivide(temp2);\n"^
"            case LDIV:\n"^
"                return temp1.solve(temp2);\n"^
"            case ERDIV:\n"^
"                return temp1.arrayRightDivide(temp2);\n"^
"        }\n"^
"    }\n"^
"    return null;\n"^
"  }\n"^
"static Matrix matrixInit(int row, int col, double [] array){\n"^
"    double [][]m=new double[row][col];\n"^
"    int arrayIndex=0;\n"^
"    for (int i=0; i<row;i++ ){\n"^
"        for (int j=0; j<col; j++){\n"^
"          m[i][j]=array[arrayIndex];\n"^
"           arrayIndex++;\n"^
"        }\n"^
"    }\n"^
"    return new Matrix(m);\n"^
"  }\n"^
"public static int getHeight(Matrix M){\n"^
"    return M.getRowDimension();\n"^
```

```
"}\n"^
"public static int getWidth(Matrix M){\n"^
"       return M.getColumnDimension();\n"^
"}\n"^
"static Matrix getSubMatrix(Matrix M, int i0, int i1, int j0, int j1){\n"^
"       int start1=0;\n"^
"       int finish1=0;\n"^
"       int start2=0;\n"^
"       int finish2=0;\n"^
"       if (i0==-1 && i1==-1){\n"^
"          start1=0;\n"^
"          finish1=M.getRowDimension();\n"^
"       }else{\n"^
"          start1=i0;\n"^
"          finish1=i1;\n"^
"       }\n"^
"       if (j0==-1 && j1==-1){\n"^
"          start2=0;\n"^
"          finish2=M.getColumnDimension();\n"^
"       }else{\n"^
"          start2=j0;\n"^
"          finish2=j1;\n"^
"       }\n"^
"       return M.getMatrix(start1, finish1, start2, finish2);\n"^
"}\n"^
"   public static String[] getFields(String line, String delimiters) {\n"^
"       java.util.StringTokenizer st = new java.util.StringTokenizer(line, delimiters);\n"^
"       int fieldCnt = st.countTokens();\n"^
"       String[] fields = new String[fieldCnt];\n"^
"       for (int i = 0; i < fieldCnt; i++) {\n"^
"          fields[i] = st.nextToken();\n"^
"       }\n"^
"       return fields;\n"^
"   }\n"^
"   public static void save(Matrix data, String name) {\n"^
"       File outputFile = new File(\".\" + File.separator + name + \".mat\");\n"^
"       Formatter output = null;\n"^
"       try {\n"^
"          output = new Formatter(outputFile);\n"^
"       } catch (Exception e) {\n"^
"          e.printStackTrace();\n"^
```

```
"       }\n"^
"                                          output.format(\"%d,%d\\n\",      data.getRowDimension(),
data.getColumnDimension());\n"^
"       for (int i = 0; i < data.getRowDimension(); i++) {\n"^
"           for (int j = 0; j < data.getColumnDimension(); j++) {\n"^
"               output.format(\"%e\", data.get(i, j));\n"^
"               if (j != data.getColumnDimension() - 1) {\n"^
"                   output.format(\"%s\", \",\");\n"^
"               }\n"^
"           }\n"^
"           output.format(\"%s\\n\", \"\");\n"^
"       }\n"^
"       output.flush();\n"^
"       output.close();\n"^
"   }\n"^
"   public static Matrix load( String name) {\n"^
"       File inputFile = new File(\".\" + File.separator + name + \".mat\");\n"^
"       Scanner input=null;\n"^
"       String lStr=\"\";\n"^
"       try {\n"^
"           input=new Scanner(inputFile);\n"^
"       } catch (Exception e) {\n"^
"          e.printStackTrace();\n"^
"       }\n"^
"       String []fields;\n"^
"       lStr=input.nextLine();\n"^
"        fields=getFields(lStr,\",\");\n"^
"       int nRows=Integer.parseInt(fields[0]);\n"^
"       int nCols=Integer.parseInt(fields[1]);\n"^
"       double [][] inputM=new double[nRows][nCols];\n"^
"       for (int i=0; i<nRows; i++){\n"^
"           lStr=input.nextLine();\n"^
"           fields=getFields(lStr,\",\");\n"^
"           for (int j=0; j<nCols; j++){\n"^
"               inputM[i][j]=Double.parseDouble(fields[j]);\n"^
"           }\n"^
"       }\n"^
"       Matrix oM=new Matrix(inputM);\n"^
"       return oM;\n"^
"   }\n"
```

```
(* initialize all variables *)
let initial_value vtype =
  if vtype = "void" then ""
  else if vtype = "int" then "0"
  else if vtype = "float" then "0.0"
  else if vtype = "string" then ""
  else if vtype = "boolean" then "false"
  else if vtype = "matrix" then "new Matrix(1,1)"
  else
    raise (Failure ("unrecognized type: " ^ vtype))

(* convert ODESL data type to Java data type *)
let java_type_of vtype =
  if vtype = "void" then "void"
  else if vtype = "int" then "Integer"
  else if vtype = "float" then "Double"
  else if vtype = "string" then "String"
  else if vtype = "boolean" then "Boolean"
  else if vtype = "matrix" then "Matrix"
  else
    raise (Failure ("unrecognized type: " ^ vtype))

let fsignature fdecl =
  if fdecl.fname = "main" then "public static void main(String[] args)"
  (* else "public static int " ^ fdecl.fname ^ "(int " ^ String.concat ", int " fdecl.formals ^ ")"
*)
  else
    let formal_string_list = List.fold_left (fun f_list formal -> (java_type_of formal.vartype ^ " "
^ formal.varname)::f_list) [] fdecl.formals
    in
    "public static " ^ java_type_of fdecl.rettype ^ " " ^ fdecl.fname ^ "(" ^ String.concat ","
(List.rev formal_string_list) ^ ")"


let fcall f =
  if f = "disp" then "disp"
  else if f = "width" then "getWidth"
  else if f = "height" then "getHeight"
  else if f = "sqrt" then "Math.sqrt"
  else if f = "abs" then "Math.abs"
  else if f = "exp" then "Math.exp"
```

```
  else if f = "log" then "Math.log"
  else if f = "log10" then "Math.log10"
  else if f = "sin" then "Math.sin"
  else if f = "cos" then "Math.cos"
  else if f = "tan" then "Math.tan"
  else if f = "asin" then "Math.asin"
  else if f = "acos" then "Math.acos"
  else if f = "atan" then "Math.atan"
  else if f = "atan2" then "Math.atan2"
  else if f = "setFormat" then "setNumFormat"
  else if f = "save" then "save"
  else if f = "load" then "load"
  else f

(* build a list of 'num' elements of name *)
let rec build_list (name, num, l) =
  if num = 0 then l
  else build_list (name, num-1, name::l)


(* returns the type of an expression*)
let rec type_of_expr symbols = function
    Literal(l) -> { varname = string_of_int l; vartype = "int" }
  | Floatlit(f) -> { varname = string_of_float f; vartype = "float" }
  | String(s) -> { varname = s; vartype = "string" }
  | Bool(s) -> { varname = s; vartype = "boolean" }
  | Id(s) ->
      (* try to find it in local table first *)
      let exists = List.exists (fun a -> if a.varname = s then true else false) symbols
      in
      if exists then
        let var = List.find (fun a -> if a.varname = s then true else false) symbols
        in
        var
      else (* then try to find it in global table *)
        let exists2 = List.exists (fun a -> if a.varname = s ^ "_global_var" then true else false)
symbols
        in
        if exists2 then
          let var = List.find (fun a -> if a.varname = s ^ "_global_var" then true else false)
symbols
```

```
        in
        { varname = s; vartype = var.vartype }
      else
        raise (Failure ("Unable to find " ^ s ^ " in symbol table"))
    (**
    let var_list = List.find_all (fun a -> if a.varname = s then true else false) symbols
    in
    if List.length var_list > 1 then
      raise (Failure ("Functions and variables cannot share the same name: "" ^ s ^ """))
    else
      List.hd var_list
    **)
  | Paren(e)->let pe = type_of_expr symbols e
    in pe
  | Uminus(e) ->
    let ue = type_of_expr symbols e
    in ue
  | Not(e) ->
    let ne = type_of_expr symbols e
    in ne
  | Binop(e1, o, e2) -> begin
    match o with
      Equal | Neq | Less | Leq | Greater | Geq ->
      { varname = "null"; vartype = "boolean" }
    | _ -> (* The rest of expr have the same type as the first operand *)
      let e = type_of_expr symbols e1
      in e
   end
  | Assign(v, e) ->
    (* try to find it in local table first *)
    let exists = List.exists (fun a -> if a.varname = v then true else false) symbols
    in
    if exists then
      let var = List.find (fun a -> if a.varname = v then true else false) symbols
      in
      var
    else (* then try to find it in global table *)
      let exists2 = List.exists (fun a -> if a.varname = v ^ "_global_var" then true else false)
symbols
      in
      if exists2 then
```

```
        let var = List.find (fun a -> if a.varname = v ^ "_global_var" then true else false)
symbols
        in
        { varname = v; vartype = var.vartype }
       else
        raise (Failure ("Unable to find " ^ v ^ " in symbol table"))
  | Call(f, el) ->
    let exists = List.exists (fun a -> if a.varname = f ^ "_global_func" then true else false)
symbols
    in
    if exists then (* if f is in our symbole table *)
     let ff = List.find (fun a -> if a.varname = f ^ "_global_func" then true else false)
symbols
      in
      { varname = f; vartype = ff.vartype }
    else if f = "width" then
     { varname = "width"; vartype = "int" }
    else if f = "getheight" then
     { varname = "height"; vartype = "int" }
    else if f = "load" then
     { varname = "load"; vartype = "matrix" }
    else if f = "inv" then
     { varname = "inverse"; vartype = "matrix" }
    else if f = "sqrt" then
     let arg = List.hd el
     in
     let e = type_of_expr symbols arg
     in
     if e.vartype = "int" || e.vartype ="float" then
      { varname = "sqrt"; vartype = "float" }
     else (* walker should block this*)
      { varname = ""; vartype = "" }
     else if f = "abs" then
       let arg = List.hd el
          in
          let e = type_of_expr symbols arg
       in
       {varname=e.varname; vartype=e.vartype}
     else if f = "exp" then
       let arg = List.hd el
          in
```

```
      let e = type_of_expr symbols arg
    in
    {varname=e.varname; vartype=e.vartype}
  else if f = "log" then
    let arg = List.hd el
      in
      let e = type_of_expr symbols arg
    in
    {varname=e.varname; vartype=e.vartype}
  else if f = "log10" then
    let arg = List.hd el
      in
      let e = type_of_expr symbols arg
    in
    {varname=e.varname; vartype=e.vartype}
  else if f = "sin" then
    let arg = List.hd el
      in
      let e = type_of_expr symbols arg
    in
    {varname=e.varname; vartype=e.vartype}
  else if f = "cos" then
    let arg = List.hd el
      in
      let e = type_of_expr symbols arg
    in
    {varname=e.varname; vartype=e.vartype}
  else if f = "tan" then
    let arg = List.hd el
      in
      let e = type_of_expr symbols arg
    in
    {varname=e.varname; vartype=e.vartype}
  else if f = "asin" then
    let arg = List.hd el
      in
      let e = type_of_expr symbols arg
    in
    {varname=e.varname; vartype=e.vartype}
  else if f = "acos" then
    let arg = List.hd el
```

```
            in
            let e = type_of_expr symbols arg
          in
          {varname=e.varname; vartype=e.vartype}
        else if f = "atan" then
          let arg = List.hd el
            in
            let e = type_of_expr symbols arg
          in
          {varname=e.varname; vartype=e.vartype}
        else if f = "atan2" then
          let arg = List.hd el
            in
            let e = type_of_expr symbols arg
          in
          {varname=e.varname; vartype=e.vartype}
        else if f = "mod" then
          let arg = List.hd el
            in
            let e = type_of_expr symbols arg
          in
          {varname=e.varname; vartype=e.vartype}
        else if f = "setFormat" then
          {varname="setFoprmat"; vartype="void"}
        else if f = "save" then
          {varname="save"; vartype="void"}
        else if f = "load" then
           {varname="load"; vartype="matrix"}
        else if f = "tostring" then
        { varname = "tostring"; vartype = "string" }
      else
        { varname = ""; vartype = "" }
    | _ -> { varname = ""; vartype = "" }

(* given variable v, find it in symbol table *)
let rec find_symbol symbols v =
  (* try to find it in local table first *)
  let exists = List.exists (fun a -> if a.varname = v then true else false) symbols
  in
  if exists then
    let var = List.find (fun a -> if a.varname = v then true else false) symbols
```

```
    in
    var
  else (* then try to find it in global table *)
    let exists2 = List.exists (fun a -> if a.varname = v ^ "_global_var" then true else false)
symbols
    in
    if exists2 then
      let var = List.find (fun a -> if a.varname = v ^ "_global_var" then true else false)
symbols
      in
      { varname = v; vartype = var.vartype }
    else
      raise (Failure ("Unable to find " ^ v ^ " in symbol table"))

let rec string_of_expr symbols = function
    Literal(l) -> string_of_int l
  | Floatlit(l) -> string_of_float l (* cast to float..Java's default type is double *)
  | String(s) -> "\"" ^ s ^ "\""
  | Bool(s) -> s
  | Id(s) -> s
  | Paren(e)-> "("^string_of_expr symbols e ^ ")"
  | Transpose(e)->string_of_expr symbols e ^".transpose()"
  | Uminus(e) -> "-(" ^ string_of_expr symbols e ^ ")"
  | Not(e) -> "!(" ^ string_of_expr symbols e ^ ")"
  | Binop(e1, o, e2) -> (* need to handle special cases here where e1 is a matrix type*)
      let e1_decl = type_of_expr symbols e1
      in
      let e2_decl = type_of_expr symbols e2
      in
      if e1_decl.vartype = "matrix" && (e2_decl.vartype="int" || e2_decl.vartype="float")
then
        let op =
          (match o with
            Add -> "OPERATION.ADD" | Sub -> "OPERATION.SUB" |
            Mul|Emul -> "OPERATION.MUL" | Div|Erdiv -> "OPERATION.DIV" | Power-
>"OPERATION.POW"|
            Ldiv|Eldiv|Equal|Neq|Less | Leq | Greater | Geq |
            And | Or ->
              raise (Failure ("For " ^ e1_decl.varname ^ " and " ^ e2_decl.varname ^", the
only arithmetic operations allowed for matrix are +, -, *, /, ./ and ^"))
          )
```

```
      in
      "matrixOperations(" ^ string_of_expr symbols e1 ^ ", " ^
                          string_of_expr symbols e2 ^ ", " ^
                          op ^ ")"
    else if e2_decl.vartype = "matrix" && (e1_decl.vartype="int" ||
e1_decl.vartype="float") then
      let op =
        (match o with
          Add -> "OPERATION.ADD" | Sub -> "OPERATION.SUB" |
          Mul|Emul -> "OPERATION.MUL" | Div|Erdiv -> "OPERATION.DIV" |
          Ldiv|Eldiv|Equal|Neq|Less | Leq | Greater | Geq |Power|
          And | Or ->
            raise (Failure ("For " ^ e1_decl.varname ^ " and " ^ e2_decl.varname ^", the
only arithmetic operations allowed for matrix are +, -, *, / and./"))
        )
      in
        "matrixOperations(" ^ string_of_expr symbols e1 ^ ", " ^
                          string_of_expr symbols e2 ^ ", " ^
                          op ^ ")"
    else if e2_decl.vartype = "matrix" && e1_decl.vartype="matrix" then
        let op =
        (match o with
          Add -> "OPERATION.ADD" | Sub -> "OPERATION.SUB" |
          Mul -> "OPERATION.MUL" | Ldiv-> "OPERATION.LDIV"|
          Emul-> "OPERATION.EMUL"| Erdiv-> "OPERATION.ERDIV"|
          Eldiv-> "OPERATION.ELDIV"|
           Equal|Neq|Less | Leq | Greater | Geq |Div|
          And | Or | Power ->
            raise (Failure ("For " ^ e1_decl.varname ^ " and " ^ e2_decl.varname ^", the
only arithmetic operations allowed for matrix are +, -, *, /, .*, ./, \\")))
      in
      "matrixOperations(" ^ string_of_expr symbols e1 ^ ", " ^
                          string_of_expr symbols e2 ^ ", " ^
                          op ^ ")"
    else
      let op =
        (match o with
          Power -> "POWER"
        | _ -> "")
      in
      if op = "POWER" then
```

```
        if (e1_decl.vartype = "int" && e2_decl.vartype = "int")
          || (e1_decl.vartype = "float" && e2_decl.vartype = "float")
          || (e1_decl.vartype = "int" && e2_decl.vartype = "float")
          || (e1_decl.vartype = "float" && e2_decl.vartype = "int")
          then
          "Math.pow(" ^ string_of_expr symbols e1 ^ ", " ^
                        string_of_expr symbols e2 ^ ")"
        else if e1_decl.vartype = "matrix" && (e2_decl.vartype = "int" ||
e2_decl.vartype="float") then
          "matrixOperations(" ^string_of_expr symbols e1 ^ ", " ^string_of_expr symbols e2 ^
","^"OPERATION.POW"^")"
        else "" (* semantics.ml should block this *)
       else
        string_of_expr symbols e1 ^ " " ^
         (match o with
              Add -> "+" | Sub -> "-" | Mul -> "*" | Div -> "/"
         | Equal -> "==" | Neq -> "!="
         | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">="
         | And -> "&&" | Or -> "||"
         | Power -> "" (* this should never match! *)
         |_->raise (Failure ("Operation not allowed for "^ string_of_expr symbols e1^ " and "
^ string_of_expr symbols e2^"."  ))
        ) ^" "^string_of_expr symbols e2
  | Assign(v, e) ->
      v ^ " = " ^ string_of_expr symbols e
  | Call(f, el) ->
    let symbols_list = build_list (symbols, List.length el, [])
    in
    if f = "mod" then
      let e1 = List.nth el 0 in
      let e2 = List.nth el 1 in
      string_of_expr symbols e1 ^ " % " ^ string_of_expr symbols e2
    else if f = "inv" then
         let e1 = List.nth el 0 in
         string_of_expr symbols e1 ^ ".inverse()"
     else if f = "tostring" then
         let e = List.nth el 0 in
        "String.valueOf(" ^ string_of_expr symbols e ^ ")"
    else
      fcall f ^ "(" ^ String.concat ", " (List.map2 string_of_expr symbols_list el) ^ ")"
  | Datatype(_) -> ""
```

```
  | Newmat(e1, e2)->
      "new Matrix(" ^ string_of_expr symbols e1 ^ ", " ^
              string_of_expr symbols e2 ^ ")"
  | AssignMatrix(id, indexList, value) ->
    let i1=List.nth indexList 1 in
    let i2=List.nth indexList 0 in
    id ^".set(" ^string_of_expr symbols i1 ^", " ^string_of_expr symbols i2 ^ ", "
^string_of_expr symbols value ^")"
  | MatrixAccess(id, indexList) ->
    let i1=List.nth indexList 1 in
    let i2=List.nth indexList 0 in
    id ^".get("^string_of_expr symbols i1^", "^ string_of_expr symbols i2^")"
  | MatrixRAccess(id, i1, i2, j1, j2) ->
    "getSubMatrix(" ^ id ^ ", " ^ (string_of_expr symbols i1) ^ ", " ^ (string_of_expr symbols
i2) ^ ", " ^ (string_of_expr symbols j1) ^ ", " ^ (string_of_expr symbols j2) ^ ")"
(*let rec string_of_stmt retname = function*)

(* convert 2D matrix to 1D matrix *)
(* e.g. 3x3 to 1x9 *)
let rec expand_matrix_list a l =
  let lr = List.rev l
  in a @ lr


let rec string_of_stmt symbols = function
    Block(stmts) ->
      (*let retname_list = build_list (retname, List.length stmts, [])*)
      let symbols_list = build_list (symbols, List.length stmts, [])
      in
      (*"{\n" ^ String.concat "" (List.map2 string_of_stmt retname_list stmts) ^ "}\n"*)
      "{\n" ^ String.concat "" (List.map2 string_of_stmt symbols_list stmts) ^ "}\n"
  | Expr(expr) -> begin
      match expr with
        (* we ignore stmts that don't have lvalue to store the result of an expr *)
        Literal(_) | Floatlit(_) | String(_) | Id(_) | Uminus(_) | Binop(_, _, _) -> ""
      | _ -> string_of_expr symbols expr ^ ";\n";
    end
  | If(e, s1, s2) ->  "if (" ^ string_of_expr symbols e ^ ")\n" ^
      string_of_stmt symbols s1 ^ "else\n" ^ string_of_stmt symbols s2
  | IfelseIf(e, s1, eif_list, s2) ->
      let rec print_elseif eif =
```

```
      "else if(" ^ string_of_expr symbols eif.elseif_expr ^ ")\n" ^
      string_of_stmt symbols eif.elseif_stmt
    in
    "if (" ^ string_of_expr symbols e ^ ")\n" ^ string_of_stmt symbols s1 ^
    (String.concat "" (List.map print_elseif eif_list)) ^ "else\n" ^ string_of_stmt symbols s2
  | For(Assigna(v, e1, e2), s) ->
    "for (" ^ v ^ " = " ^ string_of_expr symbols e1 ^ " ; " ^ v ^ " <= " ^ string_of_expr symbols
e2 ^ " ; " ^
    v ^ "++)" ^ string_of_stmt symbols s
  | For(Assignb(v, e1, e2, e3), s) -> begin
    match e2 with
      Literal(l) ->
        if l = 0 then
          raise (Failure ("increment value cannot be zero"))
        else
          "for (" ^ v ^ " = " ^ string_of_expr symbols e1 ^ " ; " ^ v ^ " <= " ^ string_of_expr
symbols e3 ^ " ; " ^
          v ^ " += " ^ string_of_expr symbols e2 ^ ")" ^ string_of_stmt symbols s
    | Floatlit(l) ->
        if l = 0.0 then
          raise (Failure ("increment value cannot be zero"))
        else
          "for (" ^ v ^ " = " ^ string_of_expr symbols e1 ^ " ; " ^ v ^ " <= " ^ string_of_expr
symbols e3 ^ " ; " ^
          v ^ " += " ^ string_of_expr symbols e2 ^ ")" ^ string_of_stmt symbols s
    | Uminus(Literal(l)) ->
        if l = 0 then
          raise (Failure ("increment value cannot be zero"))
        else
          "for (" ^ v ^ " = " ^ string_of_expr symbols e1 ^ " ; " ^ v ^ " >= " ^ string_of_expr
symbols e3 ^ " ; " ^
          v ^ " += " ^ string_of_expr symbols e2 ^ ")" ^ string_of_stmt symbols s
    | Uminus(Floatlit(l)) ->
        if l = 0.0 then
          raise (Failure ("increment value cannot be zero"))
        else
          "for (" ^ v ^ " = " ^ string_of_expr symbols e1 ^ " ; " ^ v ^ " >= " ^ string_of_expr
symbols e3 ^ " ; " ^
          v ^ " += " ^ string_of_expr symbols e2 ^ ")" ^ string_of_stmt symbols s
    | _ -> raise (Failure ("increment value must be an integer or float literal: " ^
string_of_expr symbols e2))
```

```
    end


  | While(e, s) -> "while (" ^ string_of_expr symbols e ^ ") " ^ string_of_stmt symbols s
  | MatrixInit(k, l) ->
      let elist = List.fold_left expand_matrix_list [] l
      in
      let symbols_list = build_list (symbols, List.length elist, [])
      in
      let slist = List.map2 string_of_expr symbols_list elist
      in
      let col = string_of_int (List.hd (List.map (fun subl -> List.length subl) l))
      in
      let row = string_of_int (List.length l)
      in
      k ^ " = matrixInit (" ^ row ^ ", "
                         ^ col ^ ", new double[]{" ^
                        (String.concat ", " slist) ^ "});\n"
  | Break-> "break"
  | Continue-> "continue"
let string_of_vdecl var = java_type_of var.vartype ^ " " ^ var.varname ^ " = " ^
  initial_value var.vartype ^ ";\n"

let string_of_global_vdecl var =
  "public static " ^ java_type_of var.vartype ^ " " ^ var.varname ^ " = " ^
  initial_value var.vartype ^ ";\n"

(* declare return variable *)
let retvar_decl fdecl =
  if fdecl.rettype = "void" then ""
  else java_type_of fdecl.rettype ^ " " ^ fdecl.retname ^ " = " ^
  initial_value fdecl.rettype ^ ";\n"

(* return retname *)
let retvar_return fdecl =
  if fdecl.rettype = "void" then ""
  else "return " ^ fdecl.retname ^ ";\n"

let string_of_fdecl fdecl globals =
  (* let retname_list = build_list (fdecl.retname, List.length fdecl.body, []) *)
```

```
  let symbols = globals @ fdecl.locals @ fdecl.formals @ [{ varname = fdecl.retname;
vartype = fdecl.rettype}]
  in
  let symbols_list = build_list (symbols, List.length fdecl.body, [])
  in
  fsignature fdecl ^ "\n{\n" ^
  retvar_decl fdecl ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  (* String.concat "" (List.map2 string_of_stmt retname_list fdecl.body) ^ *)
  String.concat "" (List.map2 string_of_stmt symbols_list fdecl.body) ^
  retvar_return fdecl ^
  "}\n"

(*
let rec merge_vars_funcs vars funcs =
  if List.length funcs = 0 then vars
  else
    let func = List.hd funcs
    in
    { varname = func.fname; vartype = func.rettype } :: merge_vars_funcs vars (List.tl
funcs)
*)

let string_of_program (vars, funcs) className=
  (* let globals = merge_vars_funcs vars funcs *)
  let global_funcs = List.map (fun a -> { varname = a.fname ^ "_global_func"; vartype =
a.rettype }) funcs
  in
  let global_vars = List.map (fun a -> { varname = a.varname ^ "_global_var"; vartype =
a.vartype }) vars
  in
  let globals = global_funcs @ global_vars (* concatenate global variables and global
functions to form symbols *)
  in
  let globals_list = build_list (globals, List.length funcs, [])
  in
  import_decl ^ "public class " ^className^ " {\n"^class_decl ^
  String.concat "" (List.map string_of_global_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map2 string_of_fdecl funcs globals_list) ^ "\n}\n" ^
  "\n"
```

MSCL.ml

```
(**************)
(* The Driver *)
open Printf
let usage_msg = "Usage: mscl [ -analyze ] input_file"

(* Command-line options and setters for Arg.parse *)
let analyze = ref false
let input_file = ref ""
let output_file = ref ""
let set_analyze ()   = analyze := true
let set_input_file f = input_file := f

(* Main *)
let main () =
  let desc =
    [("-analyze",   Arg.Unit(set_analyze),   "Semantic analysis and exit.")]
  in
  Arg.parse desc set_input_file usage_msg;

  if !input_file = "" then (print_endline usage_msg; exit 1);
  let dotindex=
          try (String.rindex !input_file '.') with
                  | Not_found-> String.length !input_file
                  | _->String.rindex !input_file '.'
  in
  let inputName=String.sub !input_file 0 (dotindex)
  in
  let output_file= inputName ^ ".java"
  in

  (* Syntax *)
  let input = open_in !input_file in
  let lexbuf = Lexing.from_channel input in
  let program = Parser.program Scanner.token lexbuf in
  close_in input;

  (* Semantics *)
  ignore (Semantics.string_of_program program);
  if !analyze then exit 0;
  let oc=open_out output_file in
```

```
 let java = Javawriter.string_of_program program inputName in
 fprintf oc "%s" java;
 close_out oc;
 print_endline (output_file^" has been outputted.");
  exit 0
 (* Run main *)
  let _ = try main () with Failure(s) -> (print_endline s; exit 1)
```

Make File (Make.bat)

```
ocamllex scanner.mll
ocamlyacc parser.mly
ocamlc -c ast.mli
ocamlc -c parser.mli
ocamlc -c scanner.ml
ocamlc -c parser.ml
ocamlc -c semantics.ml
ocamlc -c javawriter.ml
ocamlc -c mscl.ml
ocamlc -o mscl.exe parser.cmo scanner.cmo semantics.cmo javawriter.cmo mscl.cmo
```

Test Script (runTest.bat)

```
mscl test_math6.mscl
  javac -cp jama.jar test_math6.java
 java -cp jama.jar;. test_math6> test_math6.out

 mscl test_matrix_add.mscl
  javac -cp jama.jar test_matrix_add.java
 java -cp jama.jar;. test_matrix_add> test_matrix_add.out

 mscl test_matrix_assign.mscl
  javac -cp jama.jar test_matrix_assign.java
 java -cp jama.jar;. test_matrix_assign> test_matrix_assign.out

 mscl test_matrix_dim_inv.mscl
  javac -cp jama.jar test_matrix_dim_inv.java
 java -cp jama.jar;. test_matrix_dim_inv> test_matrix_dim_inv.out

 mscl test_matrix_div.mscl
  javac -cp jama.jar test_matrix_div.java
 java -cp jama.jar;. test_matrix_div> test_matrix_div.out
```

```
mscl test_matrix_elementOp.mscl
 javac -cp jama.jar test_matrix_elementOp.java
java -cp jama.jar;. test_matrix_elementOp> test_matrix_elementOp.out

mscl test_matrix_mult.mscl
 javac -cp jama.jar test_matrix_mult.java
java -cp jama.jar;. test_matrix_mult> test_matrix_mult.out

mscl test_matrix_range.mscl
 javac -cp jama.jar test_matrix_range.java
java -cp jama.jar;. test_matrix_range> test_matrix_range.out

mscl test_matrix_rotate.mscl
 javac -cp jama.jar test_matrix_rotate.java
java -cp jama.jar;. test_matrix_rotate> test_matrix_rotate.out

mscl test_matrix_save_load.mscl
 javac -cp jama.jar test_matrix_save_load.java
java -cp jama.jar;. test_matrix_save_load> test_matrix_save_load.out

mscl test_matrix_transpose.mscl
 javac -cp jama.jar test_matrix_transpose.java
java -cp jama.jar;. test_matrix_transpose> test_matrix_transpose.out

mscl test_minus.mscl
 javac -cp jama.jar test_minus.java
java -cp jama.jar;. test_minus> test_minus.out

mscl test_namespace1.mscl
 javac -cp jama.jar test_namespace1.java
java -cp jama.jar;. test_namespace1> test_namespace1.out

mscl test_namespace2.mscl
 javac -cp jama.jar test_namespace2.java
java -cp jama.jar;. test_namespace2> test_namespace2.out

mscl test_namespace3.mscl
 javac -cp jama.jar test_namespace3.java
java -cp jama.jar;. test_namespace3> test_namespace3.out

mscl test_namespace4.mscl
```

```
 javac -cp jama.jar test_namespace4.java
java -cp jama.jar;. test_namespace4> test_namespace4.out

mscl test_namespace5.mscl
 javac -cp jama.jar test_namespace5.java
java -cp jama.jar;. test_namespace5> test_namespace5.out

mscl test_namespace6.mscl
 javac -cp jama.jar test_namespace6.java
java -cp jama.jar;. test_namespace6> test_namespace6.out

mscl test_nestedfun.mscl
 javac -cp jama.jar test_nestedfun.java
java -cp jama.jar;. test_nestedfun> test_nestedfun.out

mscl test_recfun.mscl
  javac -cp jama.jar test_recfun.java
java -cp jama.jar;. test_recfun> test_recfun.out
```

hello.mscl

```
function main()
  disp("Hello World");
end%This is a test case for solve initial value problem using euler's method
```

odeEuler.mscl

```
%function float output=dy(float t, float y)
%       %differential equation dy=(t-y)/2
%       output=(t-y)/2;
%end

%function float output=y(float t)
%       %Exact solution of dy
%       output=3*exp(-t/2)-2+t;
%end

function float output=dy(float t, float y)
        %differential equation
        output=t^2-y;
end

function float output=y(float t)
```

```
	%Exact solution of d
	output=-exp(-t)+t^2-2*t+2;
end
function matrix E=Euler(float a, float b, float ya, int N)
	%Solve differential equation dy using Euler's method
	%y_k+1=y_k+hf(t_k, y_k) for k=0,1,2..N-1
	%a and b are the left and right end points
	%ya is the initial condition y(a)
	%N is the numbe rof steps
	%Output E, first column is abscissas and second coordinate is ordinates

	float h;


	int i;
	E=newmat(N+1,2);

	h=(b-a)/N;
	E[0,0]=a;
	for i=1:N
		E[i,0]=E[i-1,0]+h;
	end

	E[0,1]=ya;
	for i=0:N-1
		E[i+1,1]=E[i,1]+h*dy(E[i,0],E[i,1]);
	end

end


function main()
	%this function solves the differential equation dy=(t-y)/2 using euler's method and
displays the Euler solution with the exact solution and percentage error

	matrix Reuler; %Solution using euler method
	matrix Moutput; %output matrix
	float Rexact; %exact solution
	int n;
	int i;
```

```
	Reuler=Euler(0., 2., 1., 50);

	n=height(Reuler);
	disp(n);
	Moutput=newmat(n,4);
	for i=0:n-1
		Rexact=y(Reuler[i,0]);
		Moutput[i,0]=Reuler[i,0]; %t
		Moutput[i,1]=Reuler[i,1]; %Euler method solution
		Moutput[i,2]=Rexact; %exact solution
		Moutput[i,3]=(Moutput[i,2]-Moutput[i,1])/Moutput[i,2]*100.0;  %  percent
error
	end
	disp("	t	Euler	Exact	%Error");
	disp(Moutput);
	save(Moutput,"MoutputEuler");

end%This is a test case for solve initial value problem using Heun's method

function float output=dy(float t, float y)
	%differential equation dy=(t-y)/2
	output=(t-y)/2;
end

function float output=y(float t)
	%Exact solution of dy
	output=3*exp(-t/2)-2+t;
end
```

odeHuen.mscl

```
function matrix E=Heun(float a, float b, float ya, int N)
	%Solve differential equation dy using Heun's method
	%a and b are the left and right end points of the independent variable
	%ya is the initial condition y(a) of the dependent variable
	%N is the numbe rof steps
	%Output E, first column is indepnedent variable and second variable is
dependent variable

	float h;
	float k1;
	float k2;
```

```
	int i;
	E=newmat(N+1,2);

	h=(b-a)/N;
	E[0,0]=a;
	for i=1:N
		E[i,0]=E[i-1,0]+h;
	end

	E[0,1]=ya;
	for i=0:N-1
		k1=dy(E[i,0],E[i,1]);
		k2=dy(E[i+1,0], E[i,1]+h*k1);
		E[i+1,1]=E[i,1]+h/2*(k1+k2);
	end

end


function main()
	%this function solves the differential equation dy=(t-y)/2 using Heun's method and displays the solution with the exact solution and percentage error

	matrix Rheun; %Solution using Heun's method
	matrix Moutput; %output matrix
	float Rexact; %exact solution
	int n;
	int i;

	Rheun=Heun(0., 2., 1., 50);

	n=height(Rheun);
	disp(n);
	Moutput=newmat(n,4);
	for i=0:n-1
		Rexact=y(Rheun[i,0]);
		Moutput[i,0]=Rheun[i,0]; %t
		Moutput[i,1]=Rheun[i,1]; %Euler method solution
		Moutput[i,2]=Rexact; %exact solution
```

```
                Moutput[i,3]=(Moutput[i,2]-Moutput[i,1])/Moutput[i,2]*100.0;  %  percent
error
        end
        disp("      t       Euler      Exact      %Error");
        disp(Moutput);
        save(Moutput,"MoutputHeun");

end
```

odeRk4.mscl

```
%This is a test case for solving initial value problem using 4th order Runge-Kutta method

%function float output=dy(float t, float y)
%       %differential equation dy=(t-y)/2
%       output=(t-y)/2;
%end

function float output=dy(float t, float y)
        %differential equation
        output=t^2-y;
end

%function float output=y(float t)
%       %Exact solution of dy
%       output=3*exp(-t/2)-2+t;
%end

function float output=y(float t)
        %Exact solution of dy
        output=-exp(-t)+t^2-2*t+2;
end

function matrix E=RK4(float a, float b, float ya, int N)
%solves the diff eq using 4th order runge kutta method
%a is the initial value of the independent variable
%b is the end value of the indepenedent variable
%ya is the initial value of the the dependent variable
%N is the number of steps
%output is a matrix with independent variable in the first column and dependent
variable in the second column
```

```
    float h;
    float k1;
    float k2;
    float k3;
    float k4;
    int i;

    E=newmat(N+1,2);
    h=(b-a)/N;
    E[0,0]=a;

    for i=1:N
        E[i,0]=E[i-1,0]+h;
    end

    E[0,1]=ya;
    for i=0:N-1
        k1=h*dy(E[i,0],E[i,1]);
        k2=h*dy(E[i,0]+h/2., E[i,1]+k1/2.);
        k3=h*dy(E[i,0]+h/2., E[i,1]+k2/2.);
        k4=h*dy(E[i,0]+h,E[i,1]+k3);
        E[i+1,1]=E[i,1]+(k1+2*k2+2*k3+k4)/6.0;
    end

end


function main()
    %this function solves the differential equation dy using 4th order Runge Kutta method and displays the RK4 solution with the exact solution and percentage error

    matrix R; %Solution using RK4 method
    matrix Moutput; %output matrix
    float Rexact; %exact solution
    int n;
    int i;


    R=RK4(0., 2., 1., 50);

    n=height(R);
```

```
	Moutput=newmat(n,4);
	for i=0:n-1
		Rexact=y(R[i,0]);
		Moutput[i,0]=R[i,0]; %t
		Moutput[i,1]=R[i,1]; %Euler method solution
		Moutput[i,2]=Rexact; %exact solution
		Moutput[i,3]=(Moutput[i,2]-Moutput[i,1])/Moutput[i,2]*100.0;  %  percent
error
	end
	disp("	t		RK4	Exact	%Error");
	disp(Moutput);
	save(Moutput,"MoutputRK4");

end
```

odeRK4_2.mscl

```
%This is a test case for solving 2nd order differental equaion initial value problem using
4th order Runge-Kutta method

%The diff eq is: x''(t)+4x'(t)+5x(t)=0
%reformulate to the following form for solution:
%dx/dt=y, dy/dt=-5x-4y
%initial conditions: x(0)=3, y(0)=-5

%function float output=dy(float x, float y)
%	%differential equation
%	output=-5*x-4*y;
%end
function float output=dy(float x, float y)
	output=3*x+2*y;
end

function float output=dx(float x, float y)
	output=x+2*y;
end

%function float output=x(float t)
%	%Exact solution of dx
%	output=3*exp(-2*t)*cos(t)+exp(-2*t)*sin(t);
%end
```

```
function float output=x(float t)
        %Exact solution of dx
        output=4*exp(4*t)+2*exp(-t);
end

function matrix E=RK4_2(float a, float b, float ya, float xa, int N)
%solves the diff eq using 4th order runge kutta method
%a is the initial value of the independent variable
%b is the end value of the indepenedent variable
%ya is the initial value of dy/dt
%xa is the initial value of dx/dt

%N is the number of steps
%output is a matrix with independent variable in the first column and dependent
variable in the second column

        float h;
        float g1;
        float g2;
        float g3;
        float g4;
        float f1;
        float f2;
        float f3;
        float f4;
        float y0;
        float y;
        int i;

        E=newmat(N+1,2);
        h=(b-a)/N;
        E[0,0]=a;
        E[0,1]=xa;


        for i=1:N
                E[i,0]=E[i-1,0]+h;
        end

        y0=ya;
        disp(y);
```

```
        for i=0:N-1
                g1=dy(E[i,1],y0);
                f1=dx(E[i,1],y0);

                g2=dy(E[i,1]+h/2*f1, y0+h*g1/2.);
                f2=dx(E[i,1]+h/2*f1, y0+h/2.0*g1);

                g3=dy(E[i,1]+h/2*f2, y0+h*g2/2.);
                f3=dx(E[i,1]+h/2*f2, y0+h/2.0*g2);

                g4=dy(E[i,1]+h*f3,y0+h*g3);
                f4=dx(E[i,1]+h*f3, y0+h*g3);

                y=y0+h*(g1+2*g2+2*g3+g4)/6.0;




                E[i+1,1]=E[i,1]+h*(f1+2*f2+2*f3+f4)/6.0;
                y0=y;
        end

end


function main()
        %this function solves the differential equation dy using 4th order Runge Kutta
method and displays the RK4 solution with the exact solution and percentage error

        matrix R; %Solution using RK4 method
        matrix Moutput; %output matrix
        float Rexact; %exact solution
        int n;
        int i;
        setFormat("long");
        R=RK4_2(0.0, 0.2, 4.0, 6.0, 10);

        n=height(R);
        disp(n);
        Moutput=newmat(n,4);
```

```
	for i=0:n-1
		Rexact=x(R[i,0]);
		Moutput[i,0]=R[i,0]; %t
		Moutput[i,1]=R[i,1]; %Euler method solution
		Moutput[i,2]=Rexact; %exact solution
		Moutput[i,3]=(Moutput[i,2]-Moutput[i,1])/Moutput[i,2]*100.0;  %  percent
error
	end
	disp("	t	Euler	Exact	%Error");
	disp(Moutput);
	save(Moutput,"MoutputRK4_2");

end
```

odeRK4_3.mscl

%This is a test case for solving 2nd order differental equaion initial value problem using 4th order Runge-Kutta method

```
%The diff eq is: x''(t)+4x'(t)+5x(t)=0
%reformulate to the following form for solution:
%dx/dt=y, dy/dt=-5x-4y
%initial conditions: x(0)=3, y(0)=-5

function float output=dy(float x, float y)
	%differential equation
	output=-5*x-4*y;
end

function float output=dx(float x, float y)
	output=y;
end

function float output=x(float t)
	%Exact solution of dx
	output=3*exp(-2*t)*cos(t)+exp(-2*t)*sin(t);
end

%function float output=dy(float x, float y)
%	output=3*x+2*y;
%end
```

```
%function float output=dx(float x, float y)
%       output=x+2*y;
%end

%function float output=x(float t)
%       %Exact solution of dx
%       output=4*exp(4*t)+2*exp(-t);
%end

function matrix E=RK4_2(float a, float b, float ya, float xa, int N)
%solves the diff eq using 4th order runge kutta method
%a is the initial value of the independent variable
%b is the end value of the indepenedent variable
%ya is the initial value of dy/dt
%xa is the initial value of dx/dt

%N is the number of steps
%output is a matrix with independent variable in the first column and dependent
variable in the second column

        float h;
        float g1;
        float g2;
        float g3;
        float g4;
        float f1;
        float f2;
        float f3;
        float f4;
        float y0;
        float y;
        int i;

        E=newmat(N+1,2);
        h=(b-a)/N;
        E[0,0]=a;
        E[0,1]=xa;


        for i=1:N
```

```
            E[i,0]=E[i-1,0]+h;
        end

        y0=ya;
        disp(y);
        for i=0:N-1
            g1=dy(E[i,1],y0);
            f1=dx(E[i,1],y0);

            g2=dy(E[i,1]+h/2*f1, y0+h*g1/2.);
            f2=dx(E[i,1]+h/2*f1, y0+h/2.0*g1);

            g3=dy(E[i,1]+h/2*f2, y0+h*g2/2.);
            f3=dx(E[i,1]+h/2*f2, y0+h/2.0*g2);

            g4=dy(E[i,1]+h*f3,y0+h*g3);
            f4=dx(E[i,1]+h*f3, y0+h*g3);

            y=y0+h*(g1+2*g2+2*g3+g4)/6.0;


            E[i+1,1]=E[i,1]+h*(f1+2*f2+2*f3+f4)/6.0;
            y0=y;
        end

end


function main()
        %this function solves the differential equation dy using 4th order Runge Kutta method and displays the RK4 solution with the exact solution and percentage error

        matrix R; %Solution using RK4 method
        matrix Moutput; %output matrix
        float Rexact; %exact solution
        int n;
        int i;

        R=RK4_2(0.0, 5.0, -5.0, 3.0, 50);
        setFormat("short");
        n=height(R);
```

```
        disp(n);
        Moutput=newmat(n,4);
        for i=0:n-1
                Rexact=x(R[i,0]);
                Moutput[i,0]=R[i,0]; %t
                Moutput[i,1]=R[i,1]; %Euler method solution
                Moutput[i,2]=Rexact; %exact solution
                Moutput[i,3]=(Moutput[i,2]-Moutput[i,1])/Moutput[i,2]*100.0;    %    percent
error
        end
        disp("       t          RK4       Exact      %Error");
        disp(Moutput);
        save(Moutput,"MoutputRK4_2");

end
```

test_for_intsum.mscl

```
function  main()
%test for loop
  int x;
  int i;
  i=0;
  for i=10:-1:0
     x=x+1;
  end
  disp(x);
end
```

test_for_nested1.mscl

```
function main()
%test nested loop
  float i;
  float j;
  matrix a;
  a=newmat(3,3);
  for i=1.0:3.0
    for j=1.0:3.0
      disp(i+j);
    end
  end
```

```
end
```

test_for_nested2.mscl

```
function  main()
%nested for increments

  int i;
  int j;

  for i=1:1:3
    for j=1:1:3
      disp(i*j);
    end
  end


end
```

test_for_nested3.mscl

```
function  main()

%nested for increments

  float i;
  float j;

  for i=0.0:0.2:0.4
    for j=0.0:0.2:0.4
      disp(i+j);
    end
  end
end
```

test_for_nested4.mscl

```
function  main()
%nested for increments
```

```
  float i;
  float j;

  for i=0.4:-0.2:0.0
    for j=0.4:-0.2:0.0
      disp(i-j);
    end
  end


end
```

test_for_while.mscl

```
function main()
%test while with for
  int i;
  int j;
  i = 5;

  while i > 0

    i = i - 1;
    for j=0:3
        disp(i+j);
    end
  end

  disp("i="+i);

end
```

test_fun.mscl

```
% OK! call a function with return type without assinging it to other variable
function int m = test (int x, int y, int z)

    m=x+y+z;

end

function  main ()
```

```
  int x;
  int y;
  int z;
  x=1;
  y=2;
  z=3;
  disp(test(x,y,z));

end
```

test_if1.mscl

```
function main()
int x;
int y;
x=3;
y=4;
   if x==3
     x=x+1;
   else
     x=x-1;
   end
   disp(x);
end
```

test_if2.mscl

```
function main()
int x;
int y;
x=3;
y=4;
   if x==3
     if y==4
       x=x+1;
     else
       x=x-1;
     end
   else
     if x==5
       x=x-1;
     else
```

```
      end
   end
   disp(x);
end
```

test_if3.mscl

```
function  main()
int x;
int y;
x=3;
y=4;
   if x==3

   else
     if y==4
       x=x+1;
     else
       if y==4
         x=x+1;
           if y==5
             x=x+1;
           else
           end
       else
         x=x-1;
       end
     end
   end
   disp(x);
end
```

test_if4.mscl

```
function main()
int x;
int y;
x=3;
y=0;

   if x==4
     y=4;
   elseif x==5
     y=5;
```

```
   elseif x==6
     y=6;
   elseif x==3
     y=3;

   else
   end

disp(y);

end
```

test_if5.mscl

```
function  main()
int x;
int y;
x=3;
y=0;

   if x==4
     y=4;
   elseif x==5
     y=5;
   elseif x==6
     y=6;
   elseif x==3
      if y==0
         y=y+0;
      else
         y=y+3;
      end
   else
   end

disp(y);

end
```

test_if6.mscl

```
function main()
int x;
```

```
int y;
x=3;
y=0;

   if x==4
     y=4;
   elseif x==5
     y=5;
   elseif x==6
     y=6;
   elseif x==3
     y=3;
   else
      if x==4
         y=4;
      elseif x==5
         y=5;
       else
      end

   end

disp(y);

end
```

test_math1.mscl

```
function main()
%test simple addition
  float r;
  r=3.2+2.2;
  disp("r=3.2+2.2="+tostring(r));
end
```

test_math2.mscl

```
function main()
%test simple addition with negative numbers
  float r;
  r=-3.2+-2.2;
  disp("r=3.2+2.2="+tostring(r));
end
```

test_math3.mscl

```
function main()
%test floating point math with e
  float r;
  r=-3.+10.+1e3;
  disp("r=-3+10+1e3="+tostring(r));
end
```

test_math4.msc

```
function main()
%test a complex chain of operations with parenthesis
  float r;
  r=(1.0+2.)*3.0;
  r=2.^2.;
  r=1.+1.2*1.2-2/2*4.0+3^2^2+4.^(1./2.)+sqrt(9.)*(1.+2.-3./6.);
  setFormat("long");
  disp("r=");
  disp(r);
end
```

test_math5.mscl

```
function main()
%test a complex chain of operations with parenthesis, log, exponential and trig functions
  float r;
  float pi;
  pi=3.1415926536;
  r=abs(-1.0)*log(exp(2.0))*log10(10.)-(sin(pi)+cos(pi)+tan(pi));
  setFormat("shortE");
  disp("r=");
  disp(r);
end
```

test_math6.mscl

```
function main()
%testa series of trig functions and mod function, and nested sqrt with trig function.
  float r;
  float a;
  float b;
  float c;
```

```
  float pi;
  pi=3.1415926536;
  a=3.;
  b=4.;
  c=5.;
  r=acos(a/b);
  disp(r);
  r=asin(a/b);
  disp(r);
  r=atan(a/b);
  disp(r);
  r=atan2(b,a);
  disp(r);
  r=acos(mod(c,a)/c);
  disp(r);
  r=sqrt(sin(pi/2.));
  disp(r);
end
```

test_matrix_add.mscl

```
function main()
%test matrix addition
matrix m;
matrix x;
m=[1,2;3,4;5,6];
x=[1,1;2,1;3,4];
disp(m+1);
disp("");
disp(1+m);
disp("");
disp(2.0+m);
disp("");
disp(m+2.0);
disp("");
disp(m+x);
end
```

test_matrix_assign.mscl

```
function main()
%test matrix assignment and for loop
        matrix m;
```

```
        int i;
        m=newmat(1,6);
        for i=0:5
                m[0,i]=i;
        end

        for  i=0:5
                disp(m[0,i]);
        end

end
```

test_matrix_save_load.mscl

```
function main()
%test matrix save and load
        matrix m;
        matrix s;
        m=[1,0,0,1;0,1,0,1;0,0,1,1];
        disp("width of m=" + tostring(width(m)));
        disp("height of m=" + tostring(height(m)));
        s=m[0:2,0:2];
        disp(inv(s));

end
```

test_matrix_div.mscl

```
function main()
%test matrix scaler division
matrix m;
matrix x;
m=[2,2;2,2];
x=[2;2];
disp(m/1);
disp("");
disp(1/m);
disp("");
disp(2.0/m);
disp("");
disp(m/2.0);

end
```

test_matrix_elementOp.mscl

```
function main()
%test matrix element by element operations
matrix m;
matrix x;
m=[1,2,3,4];
x=[2,4,6,8];
disp(m./x);
disp("");
disp(m.\x);
disp("");
disp(m.*x);
disp("");
disp(m^2.0);
disp("");
disp(x^1);

end
```

test_matrix_rotate.mscl

```
function main()
%test matrix operations by testing a rotate vector function
        matrix v;
        pi=3.1415926;
        v=[0,0,1]; %unit vector pointing in z direction
        disp(rotatex(v,30.0)); %rotate about x axis 30 deg, display resulting rotated vector
end

function matrix v = rotatex(matrix m, float angle)
        matrix Rx;
        Rx=[1,0,0;0,cos(angle*pi/180.0),-
sin(angle*pi/180.0);0,sin(angle*pi/180.),cos(angle*pi/180.0)];
        v=m*Rx;
end
```

test_matrix_transpose.mscl

```
function main()
%test matrix transpose
matrix m;
```

```
m=[1,2;3,4;5,6];
disp(m);
disp("");
disp(m');
endfunction  main()
  int i;
  int j;
  int k;
  i = 3;
  j = ---6;

  k = - 5 - - j + -3;
  disp(k);

end
```

test_namespace1.mscl

```
%global variable name is the same with global function name
int x;
function int m=x()
  m=2;
end
function  main()
    disp(x);
    disp(x());
end
```

test_namespace2.mscl

```
%local variable is the parameter of the global function
function int x=x()
  x=2;
end
function  main()
    disp(x());
end
```

test_namespace3.mscl

```
%local variable is inside the scope of the global function
function int m=x()
 int x;
  x=2;
```

```
   m=x;
end
function  main()
     disp(x());
end
```

test_namespace4.mscl

```
%local variable is inside the scope of the global function
function int m=test()
  int x;
   x=2;
   m=x;
end
function  main()
   int test;
   test=7;
   disp(test());
   disp(test);
end
```

test_namespace5.mscl

```
int x;
function int x=test()
  x=4;
end

function main()

   disp(x);
   disp(test());

end
```

test_namespace6.mscl

```
int x;
function int m=test()
  int x;
  x=3;
  m=x;
end
function  main()
```

```
  disp(x);
  disp(test());

end
```

test_nestedfun.mscl

```
% OK -1 nested recursive
function int m = foo (int x)
   if(x > 0)
      m=foo(x-1);
   else
      m=-1;
   end
end

function int m = bar (int x)
   m=x+1;
end

function main()
   disp(foo(bar(5)));
end
```

test_recfun.mscl

```
% OK -1 nested recursive
function int m = foo (int x)
   if(x > 0)
      m=foo(x-1);
   else
      m=-1;
   end
end

function int m = bar (int x)
   m=x+1;
end

function  main()
   disp(foo(bar(foo(5))));
end
```