LAME: Linear Algebra Made Easy

David Golub
Carmine Elvezio
Ariel Deitcher
Muhammad Ali Akbar

December 20, 2010

1 Project Proposal

We propose to develop a language with built-in support for linear algebra and matrix operations. The language will provide functionality similar to MATLAB from The MathWorks, Inc. However, the syntax will be similar to C, C++, or Java.

Our language will provide four primitive data types, scalar, matrix, string, and Boolean. These data types will do as their names suggest. A scalar will hold a double-precision floating point number. A matrix will store a two-dimensional array of double-precision floating point values. Matrices will be declared using commas to separate columns and semicolons to separate rows. For example, the code

```
matrix A = {
    1, 2, 3;
    4, 5, 6;
    7, 8, 9
};
```

will correspond to the matrix

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

in the standard notation from linear algebra. Individual elements of a matrix variable will be able to be access and modified using the syntax `A[i, j]` for the element in the ith row and the jth column. Keywords will be provided to obtain the dimensions of a matrix.

Operators will be provided for addition, subtraction, multiplication, division, and exponentiation. The operators will be interpreted appropriately for various

combinations of operand types, as long as they are mathematically meaningful. Combinations of operand types that are not mathematically meaningful, such as division of two matrices, will yield a compiler error. When matrix operations are performed, a check will be done at runtime to ensure that the dimensions are compatible. If the check fails, a runtime error will be thrown.

The language will follow the imperative paradigm and will provide constructs for variable assignment, decisions, loops, and basic I/O. Programs will be translated to an intermediate code and then to C++ code using a custom library. The C++ code can then be compiled to native code.

2 Tutorial

This is a beginner's tutorial to LAME. LAME is a C-like programming language for linear algebra. It allows a mathematician to implement algorithms involving matrix operations with a very short learning curve.

This tutorial is divided in to three parts. First, we describe the steps involved in compiling and running a simple "Hello, World!" program. Then, we describe how to perform basic matrix operations. Finally, we explain the construction of a program that implements the algorithm to solve a system of linear equations.

2.1 Getting Started

We start by writing a very basic program that prints "Hello, World!"

2.1.1 "Hello, World!" Program

Type the following program in a file names `hello.lam`.

```
print "Hello, World!";
```

The `print` keyword takes a string as an argument and prints that string to the output console. Each statement in LAME ends with a semicolon.

2.1.2 Compiling and Running the Program

You should have a C++ compiler on your machine. The `lame.exe` executable, the `compiler.bat` script, the `lame.h` and `matrix.h` header files should be present in the working directory for the compilation.

Windows: First, we give instructions for a Windows machine with Visual C++. First set up the environment by running `vcvars32.bat` file from Visual Studio's common folder in the installation path. Alternatively, you can use the Visual Studio Command Prompt which automatically sets the environment variables for the compiler.

Now run the following commands to compile and run the program:

```
> lame.exe < hello.lam > program.cpp
> cl.exe /EHsc program.cpp
> program.exe
```

Alternatively, you can use the provided batch script by using following command:

```
> compiler.bat hello.lam
> program.exe
```

Linux: Run the following commands to compile and run the program:

```
# ./lame < hello.lam > program.cpp
# g++ program.cpp -o program
# ./program
```

If running the program prints "Hello, World!" on a separate line on the screen, and you are ready to go to next part of the tutorial.

2.2 Basic Matrix Operations

Let's write a program that declares a matrix with initial values and prints the matrix:

```
matrix A = { 1, 2; 3, 4 };
print "A = \n" + A;
```

Let's declare another matrix `B` and add it to the matrix `A`:

```
matrix B = { 4, 3; 2, 1 };
matrix C = A + B;
print "A + B = \n" + C;
```

Subtraction and multiplication of matrices are done in similar manner. Let's multiply the matrix by a scalar value:

```
matrix D = 2 * A;
print "2 * A = \n" + D;
```

Let's change the value of an element in A and print the value of an element in B:

```
A[0, 1] = 20;
print "B[1, 1] = " + B[1, 1];
```

The following `if` statement prints "True" and the `while` loop prints all elements of matrix `A`:

```
if(A[0, 0] == 1) {
    print "True";
} else {
    print "False";
}
scalar i = 0;
scalar j = 0;
while(i < 2) {
    while(j < 2) {
        print A[i, j] + "\n";
        j = j + 1;
    }
    i = i + 1;
}
```

2.3 Case Study: Solving Linear System of Equations

Let us implement an algorithm using LAME to solve a system of simultaneous linear equations. The equations are

$$3x_1 + x_2 = 3$$

and

$$9x_1 + 4x_2 = 6.$$

We use the following algorithm to solve this problem:

$$A = \begin{bmatrix} 3 & 1 \\ 9 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 3 \\ 6 \end{bmatrix}$$

$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = A^{-1}B$$

$$A^{-1} = \frac{1}{|A|}\text{Adj}(A) = \frac{1}{A_{0,0}A_{1,1} - A_{0,1}A_{1,0}} \begin{bmatrix} A_{1,1} & -A_{0,1} \\ -A_{1,0} & A_{0,0} \end{bmatrix}$$

This algorithm has been implemented in LAME as follows:

```
matrix A = { 3, 1; 9, 4 };
matrix B = { 3; 6 };
matrix X;
print "\nSolving system of simultaneous linear equations:\n";
```

```
print A[0,0] + " x1 + " + A[0,1] + " x2 = " + B[0] + "\n";
print A[1,0] + " x1 + " + A[1,1] + " x2 = " + B[1] + "\n";
print "\nA = \n" + A + "\n";
print "\nB = \n" + B + "\n";
scalar det_of_A = A[0,0]*A[1,1] - A[0,1]*A[1,0];
print "\nDeterminant(A) = " + det_of_A + "\n";
if(det_of_A != 0) {
    matrix inv_of_A;
    inv_of_A [0,0] = A[1,1];
    inv_of_A [0,1] = -1*A[0,1];
    inv_of_A [1,0] = -1*A[1,0];
    inv_of_A [1,1] = A[0,0];
    inv_of_A = inv_of_A / det_of_A;
    X = inv_of_A * B;
    print "\nInverse(A) = \n" + inv_of_A + "\n";
    print "X = Inverse(A) * B = \n" + X + "\n";
    print "Solution:\n";
    print "x1 = " + X[0] + "\n";
    print "x2 = " + X[1] + "\n";
} else {
    print "A is singular and its inverse doesn't exist.\n";
}
```

The program prints out the following output:

```
Solving system of simultaneous linear equations:
3 x1 + 1 x2 = 3
9 x1 + 4 x2 = 6

A =
3 1
9 4

B =
3
6

Determinant(A) = 3

Inverse(A) =
1.33333 -0.333333
-3 1

X = Inverse(A) * B =
2
-3
```

```
Solution:
x1 = 2
x2 = -3
```

3 Language Reference Manual

3.1 Definition of a Program

A program is a sequence of variable declarations and statements.

3.2 Variable Declarations

3.2.1 Data Types

The following data types represent the complete listing of primitives available to a programmer of LAME.

- **Scalar:** This is the equivalent of a double precision floating point number in most languages. The value is 64 bit and signed. The range is as given by the IEEE floating point standard.
- **Matrix:** Matrices in LAME are represented as dynamically sized 2-dimensional arrays.
  - Elements of the matrices can be composed of either scalar values or of other matrices.
  - Matrix size is denoted using the following notation:
    - *identifier* `[`m`, `n`]`
    - The m value corresponds to the height of the matrix and the n value corresponds to the width.
  - Random access of matrix elements is handled using the following notation:
    - *identifier* `[`x`, `y`]`
    - The x value represents the row and y value represents the column.
  - Each dimension of the array uses zero-based indexing and indices must be positive.
  - Matrix literals are created by enclosing rows in curly braces.
    - Elements are separated by commas, whereas rows are separated by a semicolon. There is no semicolon needed for the final element of the matrix.
    - Every row of the matrix literal must contain the same number of elements (corresponding to the width of the matrix).

        * Whitespace is not considered in the usage of matrix literals.
    - Vectors are represented in LAME by creating a single dimensional matrix (a column vector using dimensions $n \times 1$).
    - It is possible to determine the size of a dimension of a matrix using the `size_`*`dimension`* keyword.
        * There are two variants of the keyword:
            · `size_rows` *`matrixname`*, returns the number of rows of matrixname
            · `size_cols` *`matrixname`*, returns the number of columns of matrixname

- **Boolean:** Accepts values of true and false which can be used with logical operators. Scalars cannot be implicitly converted to booleans.
- **String:** String values are collections of ASCII characters enclosed in double quotes.
    - Literals can be concatenated to other literals.
    - When attempting to print values of other data types, implicit conversion to string type occurs.

3.2.2 Identifiers

- Identifiers must begin with a letter (uppercase or lowercase), and may contain letters, digits, and underscores.

3.2.3 Scope

- Variables have global scope.
- Variables declared can be reassigned but their memory allocation will remain throughout the duration of the program.

3.2.4 Variable Declaration

Declaration of variables follow this general format:

- Without initialization
    - *`datatype identifier`*`;`
- With initialization
    - *`datatype identifier = expression`*`;`
- Type-specific initialization format:
    - **Scalar:**

    - `scalar name = 9.8165;`
- **Matrix:**
    - `matrix name = {1, 2, 3; 4, 5, 6; 7, 8, 9};`
        - Setting initial values is optional. If not initialized, the matrix is created as a zero matrix of size 1×1.
    - This corresponds to a 3×3 matrix.
    - Row elements are separated by a comma, whereas column elements are separated by a semicolon.
- **Boolean:**
    - `boolean name = true;`
    - Initial values must be either true or false.
- **String:**
    - `string name = "hello";`

3.3 Statements

3.3.1 Assignment Statements

Assignment statements in LAME are of the form

lvalue `=` *expression* `;`

The expression is evaluated and the lvalue is set to equal its value. The lvalue may be either a variable or a matrix/vector element access.

Example:

```
y = (x + 5) * 7;
A[1, 5] = 8;
```

3.3.2 Redimensioning of Matrices

The `dim` statement allows for the redimensioning of matrices. It is of the form

`dim` *matrixname* `[`*rows* `,` *cols*`];`

where both *rows* and *cols* must be integers greater than 0, or an exception is raised.

In the event that either *rows* or *cols* is **less** than the current respective sizes of the rows and/or columns, the matrix is truncated accordingly. In the event that either *rows* or *cols* is **greater** than the current respective sizes of the rows and/or columns, the appropriate number of rows and/or columns are appended to the matrix, with each newly appended entry initialized to 0.

Example:

```
dim A[3, 2];
```

3.3.3 If Statements

The `if` statement is of the form

`if(`*`condition`*`)` *`statement`*

where the statement following the `if` is only executed if the condition evaluates to true. Otherwise execution continues following the statement block.

The `else` statement, which cannot be used unless it immediately follows an `if` statement section, is of the form

`else` *`statement`*

where the statement block following the `else` is only executed if the preceding `if` evaluates to false.

Example:

```
if(A < B)
    print A;
else
    print B;
```

3.3.4 While Loops

The `while` statement is an iterative loop of the form

`while(`*`condition`*`)` *`statement`*

where the expression must evaluate to a boolean true or false. If the expression is true, the statement is executed. The expression is then re-evaluted, and the statement is executed so long as the condition is true.

Example:

```
while(A[1] < B[1])
    A[1] = A[1] + 1;
```

3.3.5 Block Statements

A block statements is a list of statements enclosed in curly braces. It is generally used to create an if statement or while loop with multiple statements that are executed conditionally or repeatedly.

Example:

```
if(x >= 5) {
    y = x + 7;
    z = 3 * y;
}
```

3.3.6 Basic I/O

Printing to standard out is done using the `print` keyword, followed by the expression to print.

Example:

```
print A;
```

3.4 Expressions

3.4.1 Literals

There are four types of literals in LAME: scalar literals, matrix literals, string literals, and Boolean literals. A scalar literal takes the form of a nonempty sequence of digits, optionally followed by a decimal point and a second nonempty sequence of digits, optionally followed by the letter E in either capital or lowercase and a nonempty sequence of digits indicating an exponent. If it is present, the sequence of digits indicating the exponent may be preceded by a minus sign to indicate that the exponent is negative. The following are examples of scalar literals:

```
5
7.3
7e-5
1.1e7
```

Matrix literals are composed of a sequence of rows separated by semicolons and enclosed in curly braces. Each row consists of a sequence of scalar literals separated by commas. All of the rows in a matrix literal must have the same number of scalar literals in them. The following are examples of matrix literals:

```
{
    1, 0, 0;
    0, 1, 0;
    0, 0, 1
}

{ 1.5, 0.5; 0.5, 1.5 }
```

A string literal is composed of a quotation mark, followed by a possibly empty sequence of string characters and/or escape sequences, followed by another quotation mark. A string character is any character except a quotation mark, backslash, or newline. An escape sequence is a backslash followed by either a quotation mark, a backslash, the lowercase letter N, or the lowercase letter T. These escape sequences denote the presence of a quotation mark, backslash, newline, or tab, respectively, in the string. The following are examples of string literals:

```
"Hello, World!"
"Say, \"Hello!\""
"First line\nSecond line"
"C:\\FOLDER\\FILENAME.EXT"
```

There are only two possible Boolean literal values, which are denoted by the keywords `true` and `false`.

3.4.2 Unary Arithmetic Operators

Two unary arithmetic operators are provided: negation and transposition. Negation is a prefix operator indicated by a minus sign and has the standard mathematical meaning. It is valid on scalars and matrices. Attempting to negate a string value will yield a compiler error. Transposition is a postfix operator indicated by an apostrophe. It is valid on matrices and scalars. On matrices, it has the standard mathematical meaning. On scalars, it has no effect. Transposing a scalar yields that same scalar. Attempting to transpose a string will yield a compiler error.

3.4.3 Binary Arithmetic Operators

LAME provides five binary arithmetic operators: addition, subtraction, multiplication, division, and exponentiation. These operators are denoted by the plus sign, minus sign, asterisk, slash, and caret, respectively. All five operators are infix operators and can be applied to scalars and have the standard mathematical meaning. Addition, subtraction, and multiplication can be applied to pairs of matrices and have the standard mathematical meaning. A runtime check will be performed to ensure that the two matrices are of sizes such that the operations can actually be performed. Multiplication can be applied to a matrix and a scalar and has the standard mathematical meaning. Division can be applied to a matrix and a scalar and has the effect of multiplying the matrix by the reciprocal of the scalar. The matrix must be the first operand and the scalar must be the second. Exponentiation can be applied to a matrix and a scalar and has the effect of multiplying the matrix by itself a number of times given by the scalar. The matrix must be the first operand and the scalar must be the second. A runtime check will be performed to ensure that the matrix is square. The addition operator can be applied to strings and in this context represents string concatenation. All other combinations of operand data types will yield a compiler error.

3.4.4 Relational Operators

LAME provides six relational operators: less than, greater than, less than or equal to, greater than or equal to, equal to, and not equal to. These operators are denoted by `<`, `>`, `<=`, `>=`, `==`, and `!=`, respectively. The last two of these operators are valid on all data types. The first four are valid on scalars and

strings only. Attempting to use them on matrices or Boolean values will yield a compiler error. All six relational operators produce a Boolean value indicating whether the corresponding relation holds.

3.4.5 Logical Operators

LAME provides three logical operators: logical and, logical or, and logical not. These operators are denoted by `&&`, `||`, and `!`, respectively. The first two are binary operators that take a pair of Boolean values and produce a Boolean value giving the result of the logical operation. Logical not is a unary prefix operator that takes a single Boolean value and produces a Boolean value giving its logical negation. All three logical operators will yield a compiler error if they are used on scalars, matrices, or strings.

3.4.6 Operator Precedence

The operator precedence and associativity for LAME is as given in the following table. Operators of the highest precedence are at the top of the table.

Operators	Associativity
Transposition	Not applicable
Negation	Not applicable
Logical not	Not applicable
Exponentiation	Left-associative
Multiplication, division	Left-associative
Addition, subtraction	Left-associative
Less than, greater than, less than or equal to, greater than or equal to	Left-associative
Equal to, not equal to	Non-associative
Logical and	Left-associative
Logical or	Left-associative

3.4.7 Matrix Element Access

The syntax of a matrix element access is as follows:

```
A[i,j]
```

This represents the matrix element A_{ij} where A is the name of the matrix, i is the row number and j is the column number.

The behavior of the matrix element access is as follows:

- For the first row, $i = 0$; for second row, $i = 1$; and so on. Similarly, for first column, $j = 0$; for second column, $j = 1$; and so on. Formally, the element access requires two scalars from set of natural numbers (including zero) i.e. $i, j \in N_0 = \{0, 1, 2, ...\}$.
- The matrix element access `A[i,j]` can be an lvalue. For example,

```
A[i,j] = 2;
```

assigns numeric value 2 to the matrix element A_{ij};

- The matrix element access `A[i,j]` returns the numeric value of the matrix element A_{ij}. For example,

  ```
  B[i,j] = A[i,j] + 2;
  ```

  adds 2 to the numeric value of A_{ij} and assigns it to the matrix element B_{ij}.

3.4.8 Vector Element Access

There is no vector type in LAME. However, the programmer is allowed to perform vector element access on a matrix. The syntax of a vector element access is as follows:

```
V[i]
```

This represents the element $V_{i,0}$ where V is the name of the matrix, i is the row number. The behavior of the Vector element access is as follows:

- For first element, $i = 0$; for second element, $i = 1$; and so on. Formally, the vector element access requires a scalar literal from set of natural numbers (including zero), i.e. $i \in N_0 = \{0, 1, 2, ...\}$.
- The vector element access `V[i]` can be an lvalue. For example,

  ```
  V[i] = 2;
  ```

  assigns numeric value 2 to the element $V_{i,0}$;

- The vector element access `V[i]` returns the numeric value of the element $V_{i,0}$. For example,

  ```
  X[i] = V[i] + 2;
  ```

  adds 2 to the numeric value of element $V_{i,0}$ and assigns it to the vector element $X_{i,0}$;

3.4.9 Implicit Casting

In `LAME`, implicit casting is supported for the print statement and the string concatenation operator. Although the print command is for the string data type, the user can call the print command on scalars and matrices. Similarly, string concatenation operator (+) when applied to 'a string and a scalar' or 'a string and a matrix' results in implicit casting of the scalar or matrix to string literal before actual concatenation is done. Implicit conversion to string data type takes place as specified below:

Scalar Literal to String Literal

A scalar literal is converted to a string literal in the standard sense. As an example, consider the statements

```
print 5;
x = 102;
print x;
print "Value of x is " + x;
```

The constant scalar value `5` is converted to string literal `"5"`, while the third statement results in implicit casting of `102` to string `"102"`. In the fourth statement, scalar x is implicitly casted to string literal `"102"`, then concatenated with the other string literal and then passed to print command. As a result, the print keyword sees this string literal as its operand: `"Value of x is 102"`.

Matrix Literal to String Literal

When a matrix is passed to the print command or a string concatenation operator along-with a string literal as the other operand, it is implicitly casted to a string literal with a specified format that is specified in the rules below.

- **Matrix Elements:** The matrix elements are all scalar and follow the rules specified in the previous paragraph about implicit casting from scalar literals to string literals.
- **Rows:** When the matrix is converted to a string literal, each row is separated by a newline (\n) character.
- **Columns:** When the matrix is converted to a string literal, each row is separated by a tab (\t) character.

As an example of implicit conversion from matrix to string literal,

```
A = { 1, 0, 0; 0, 1, 0; 0, 0, 1 };
print A;
print "A =\n" + A
```

is converted to string literal

```
"1\t0\t0\n0\t1\t0\n0\t0\t1\n"
```

and printed as

```
1    0    0
0    1    0
0    0    1
```

in first print statement, and printed as

```
A =
1    0    0
0    1    0
0    0    1
```

in second print statement.

3.5 Keyword List

The following are keywords in LAME and therefore cannot be used as identifiers:

```
boolean
else
false
if
matrix
print
scalar
size_cols
size_rows
string
true
while
```

4 Project Plan

4.1 Planning Process

By setting internal deadlines and ensuring biweekly meetings and close contact, we hope to finish the project by the last day of classes, so as to begin documentation during reading week. Each team member will have their responsibilities clearly defined so that development can be easily divided among team members. This will also allow for maximum independence whlile still ensuring cohesion.

4.2 Style Guide

All indentation of OCaml code was done using spaces. The `let` statement was formatted such that the section following the `in` keyword was indented four spaces. For example:

```
let var1 = expr1
and var2 = expr2
in
    expr3
```

The `match` statement was formatted such that the first case is indented four spaces and the subsequent cases are each indented two spaces such that the case expressions are aligned. The arrows should all be aligned one space beyond the longest case expression. For example:

```
match m with
    Case1 x      -> expr1
  | Case2 (x, y) -> expr2
```

The `if` statement was formatted such that the then- and else-clauses are each indented four spaces:

```
if condition then
    expr1
else
    expr2
```

Expressions were formatted such that binary operators are surrounded by a single space on each side. Single spaces were used to separate functions from their arguments and arguments from each other. For example:

```
f x (y + 1) + 10
```

4.3 Project Timeline

Milestone	Estimated Date
Language Whitepaper and Core Features Defined	09-07-2010
Begin LRM	10-01-2010
Finish LRM	11-01-2010
Fully Outline Grammar	11-01-2010
Complete Parser	11-10-2010
Outline iLAME (LAME intermediate code)	11-15-2010
Finish Intermediate Code	12-01-2010
Finish Code Generation	12-10-2010
Begin Final Testing	12-14-2010
Complete Project Deliverables	12-20-2010

4.4 Roles and Responsibilities

The fundamental delineations of responsibilities among team members are as follows:

Name	Responsibility
David Golub	Semantic Checking and Intermediate Code Generation
Carmine Elvezio	Parser Implementation/Unit and Final Testing
Muhammad Akbar	C++ Code Generation/C++ Linear Algebra Library
Ariel Deitcher	Planning/Documentation/Final Presentation

4.5 Development Environment

The following development environments will be used:

- OCaml
- Visual C++
- Microsoft Windows SDK

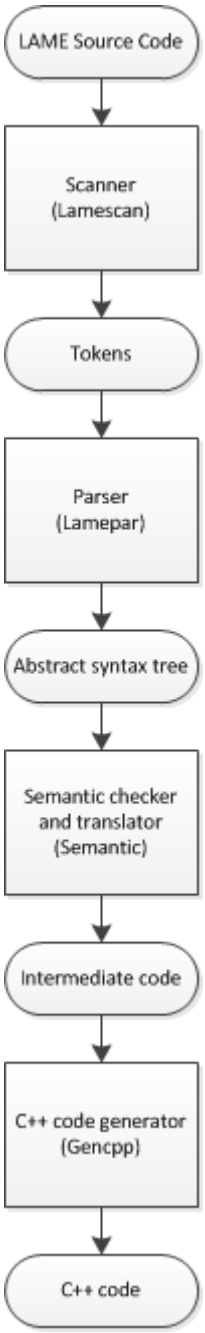
The following languages will be used:

- OCaml
- C++
- iLAME (LAME intermediate code)

4.6 Project Log

Milestone	Date
Begun LRM	10-08-2010
Grammar Finished	10-29-2010
LRM Finished	11-02-2010
Outlined iLAME	11-17-2010
Finished Semantic Checking	11-27-2010
Finished iLAME Generation	12-14-2010
Begin Regression Testing	12-15-2010
Created Makefile	12-15-2010
C++ Code Generation Completed	12-16-2010
Final Presentation	12-21-2010

5 Architectural Design



The LAME compiler is divided into four main modules, as shown in the diagram above. Each of these modules is described in one of the following sections.

5.1 Scannar

The scanner, implemented in `lamescan.mll` and the source files generated from it, divides a LAME program into tokens. This module was implemented collectively during a group meeting.

5.2 Parser

The parser, implemented in `lamepar.mly` and the source files generated from it, takes the sequence of tokens generated by the scanner and produces an abstract syntax tree from it. The format of the abstract syntax tree is given by the data structures in `ast.mli`. The original grammar was written collectively during a group meeting. The semantic actions to produce the abstract syntax tree were written by Carmine Elvezio.

5.3 Intermediate Code Generator

The intermediate code generator, implemented in `semantic.ml`, performs semantic checks on the abstract syntax tree to detect errors such as references to undeclared variables and type mismatches. It then produces an intermediate representation of the program, following the data structure given in `icode.mli`, that is guaranteed to be correct. The intermediate code is a register-based language that allows for a potentially infinite number of registers that will be mapped into variables in the generated C++ code. This module was written by David Golub.

5.4 C++ Code Generator

The C++ code generator takes as input the intermediate code and produces a semantically equivalent C++ program. Each intermediate instruction corresponds to a single line of C++ code. The C++ program uses the custom libraries defined in `matrix.h` and `lame.h`. The C++ code generator and associated libraries were written by Muhammad Ali Akbar.

6 Testing

6.1 Goals

The testing procedure for the development of LAME was planned out at the beginning to coincide with each development milestone. It was out intention to structure the testing protocol in a way that each developer would be able to internalize the exact testing method whenever unit testing was needed. At each milestone, testing was to be done on each component to ensure proper

function. At each phase of development, it was expected that tests in previous milestones would need to have been completed successfully and that new tests would incorporates elements tested in previous tests.

6.2 Hypothesis

By testing each unit using small examples as development progresses, integration and final unit testing can be completed with minimal issue.

6.3 Methods

6.3.1 Integration/Regression Testing

As each piece of code is integrated, regression testing is performed to guarantee that that the compiler can proceed successfully to the level of the integrated component. This would integrate all components that need to enter the OCaml compiler (AST, parser, intermediate code generator), and the C++ compiler.

6.3.2 Final Testing

At this point, full testing is performed of implemented assets of the language and development environment. This includes preparation of the environment on end-user machines and runtime/output analysis of complex examples.

6.4 Tools

The OCaml and C++ compilers were used to check for errors in the compilation of a program. However, as simple examples could be used to completely check our language, results were examined by hand.

6.5 Implementation

6.5.1 Integration/Regression Testing

As each portion of code, written by a different member, was integrated, different test applications were used to observe the output of the compiler as it operated. Minimal changes were necessary as few errors were encountered. The grammar was fully tested to ensure the appropriate intermediate code was created. Additionally, the scanner and parser were tested to ensure only correct abstract syntax trees were passed to the intermediate code generator.

6.5.2 Final Testing

At the last stage, we tested by attempting to write full programs that would combine different aspects of the language in a single runtime. The tests were run by different members in order to ensure the environments portability. There was full agreement in the team as to the expected performance and output and that was achieved. Automation procedures were used to check the full suite

of test applications. The automation was handled by a makefile and strict file location requirements.

6.6 Testing Responsibilities

The initial testing plan was designed by the entire team during the project planning session held at the beginning of the project. The `ocamlyacc` output was tested by all members of the team. The AST, parser, and scanner code set were tested by David Golub and Carmine Elvezio. The intermediate code generator was tested by David Golub. The C++ code generator was tested by Muhammad Ali Akbar and Carmine Elvezio. The integration testing was led by David Golub with assistance by Carmine Elvezio and Muhammad Ali Akbar. The final testing was handled by all members of the team with test applications submitted throughout the last phase.

6.7 Automation, Test Suites, and Test Cases

Automation was handled using a makefile that ran the test suite that had been prepared. The test cases were chosen in order to test specific aspects of the grammar in simple cases where errors can be clearly determined and easily solved. Larger test cases, such as `tutorial.lam`, were used in order to test algorithm implementations in our language. The test cases were collected in a single folder that represented the suite and the makefile ran the automated testing procedure. Output can be checked against the `check` directory using a utility such as `windiff`.

7 Lessons Learned

7.1 David

After writing the rough draft of our language reference manual, I realized that we should have established style guidelines for our written documents. This would have saved a great deal of time on editing. Also, I realized that the number of temporary variables generated could have been reduced by having the intermediate code generator produce either a temporary or a constant for an expression instead of a temporary for every expression.

7.2 Carmine

Defining a formal set of interface guidelines is very important and we should have prepared a document at the start. Additionally, a written version of our project plan should have been prepared in order to allow for greater organization; however, the verbal agreement between members did prove more than sufficient to allow for timely completion of the assignment.

7.3 Ariel

The following are my takeaways from this project:

- The huge importance of interfaces. Nothing is worse than working on something and then hearing your teamate say "But that's not what I was expecting!" This extends to agreeing on style guides, so that documents can be done once, and only once.
- Which naturally leads to the importance of clear division of labor. In particular each person should focus their efforts on their strengths and what they bring to the project
- Of course, none of this is possible without constant communication, especially in-person (rather than phone or even Skype) meetings.

Conclusion: Leave your ego at the door, be flexible with your personal views and communicate!

7.4 Akbar

- It would have been beneficial to have formally defined a set of interface functions before starting implementation. I had to take intermediate code as input and generate C++ code. I expected an intermediate code in the form of a text string as input. The intermediate code was being generated as a data type, which really simplified the processing, but it would have saved time spent on alternate implementation if I had confirmed my assumption earlier.
- When something can go wrong, it will go wrong. So keep pushing your changes to repository periodically

Appendix: Source Code

ast.mli

```
type row = float list

type matrix = row list

type lvalue =
    Ident of string
  | VAccess of string * expr
  | MAccess of string * expr * expr

and expr =
    NumLit of float
  | MatLit of matrix
```

```
  | StrLit of string
  | BoolLit of bool
  | LValue of lvalue
  | Plus of expr * expr
  | Minus of expr * expr
  | Times of expr * expr
  | Divide of expr * expr
  | Power of expr * expr
  | Eq of expr * expr
  | Neq of expr * expr
  | Lt of expr * expr
  | Gt of expr * expr
  | Le of expr * expr
  | Ge of expr * expr
  | Not of expr
  | And of expr * expr
  | Or of expr * expr
  | Trans of expr
  | SizeRows of expr
  | SizeCols of expr

type datatype =
    Scalar
  | String
  | Matrix
  | Boolean

type stmt =
    Assign of lvalue * expr
  | If of expr * stmt
  | IfElse of expr * stmt * stmt
  | While of expr * stmt
  | Print of expr
  | Dim of string * expr * expr
  | Decl of datatype * string
  | DeclInit of datatype * string * expr
  | StmtList of stmt list

type prgm = stmt list
```

gencpp.ml

```
open Icode
open Vars
```

```
let gencpp_init =
    "#include <iostream>\n#include \"lame.h\"\n\nusing namespace std;\n\nint main(void)\n{\n

let gencpp_end =
    "\treturn 0;\n}\n"

let string_of_bool bv =
    match bv with
        true   -> "TRUE"
      | false  -> "FALSE"

let gencpp_scalval sv vt =
    match sv with
        SLit n -> string_of_float n
      | SVar n -> get_variable_name n vt

let gencpp_boolval bv vt =
    match bv with
        BLit n -> string_of_bool n
      | BVar n -> get_variable_name n vt

let gencpp i vt =
    match i with
        SCAL (lhs, rhs)        -> let name = get_variable_name lhs vt
                                  and cppexp = gencpp_scalval rhs vt
                                  in
                                      name ^ " = " ^ cppexp ^ ";\n"
      | BOOL (lhs, rhs)        -> let name = get_variable_name lhs vt
                                  and cppexp = gencpp_boolval rhs vt
                                  in
                                      name ^ " = " ^ cppexp ^ ";\n"
      | STR (lhs, rhs)         -> let name = get_variable_name lhs vt
                                  and cppexp = rhs
                                  in
                                      name ^ " = " ^ cppexp ^ ";\n"
      | ADD (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                                  and cppexp1 = gencpp_scalval rhs1 vt
                                  and cppexp2 = gencpp_scalval rhs2 vt
                                  in
                                      name ^ " = " ^ cppexp1 ^ " + " ^ cppexp2 ^ ";\n"
      | SUB (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                                  and cppexp1 = gencpp_scalval rhs1 vt
                                  and cppexp2 = gencpp_scalval rhs2 vt
                                  in
                                      name ^ " = " ^ cppexp1 ^ " - " ^ cppexp2 ^ ";\n"
      | MUL (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
```

```
                                and cppexp1 = gencpp_scalval rhs1 vt
                                and cppexp2 = gencpp_scalval rhs2 vt
                                in
                                    name ^ " = " ^ cppexp1 ^ " * " ^ cppexp2 ^ ";\n"
    | DIV (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                                and cppexp1 = gencpp_scalval rhs1 vt
                                and cppexp2 = gencpp_scalval rhs2 vt
                                in
                                    name ^ " = " ^ cppexp1 ^ " / " ^ cppexp2 ^ ";\n"
    | POW (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                                and cppexp1 = gencpp_scalval rhs1 vt
                                and cppexp2 = gencpp_scalval rhs2 vt
                                in
                                    name ^ " = pow(" ^ cppexp1 ^ ", " ^ cppexp2 ^ ");\n"
    | MADD (lhs, rhs1, rhs2) -> let name = get_variable_name lhs vt
                                and cppexp1 = get_variable_name rhs1 vt
                                and cppexp2 = get_variable_name rhs2 vt
                                in
                                    name ^ " = " ^ cppexp1 ^ " + " ^ cppexp2 ^ ";\n"
    | MSUB (lhs, rhs1, rhs2) -> let name = get_variable_name lhs vt
                                and cppexp1 = get_variable_name rhs1 vt
                                and cppexp2 = get_variable_name rhs2 vt
                                in
                                    name ^ " = " ^ cppexp1 ^ " - " ^ cppexp2 ^ ";\n"
    | MMUL (lhs, rhs1, rhs2) -> let name = get_variable_name lhs vt
                                and cppexp1 = get_variable_name rhs1 vt
                                and cppexp2 = get_variable_name rhs2 vt
                                in
                                    name ^ " = " ^ cppexp1 ^ " * " ^ cppexp2 ^ ";\n"
    | SMUL (lhs, rhs1, rhs2) -> let name = get_variable_name lhs vt
                                and cppexp1 = get_variable_name rhs1 vt
                                and cppexp2 = gencpp_scalval rhs2 vt
                                in
                                    "LAMEScalarMul(" ^ name ^ ", " ^ cppexp1 ^ ", " ^ cppe
    | MPOW (lhs, rhs1, rhs2) -> let name = get_variable_name lhs vt
                                and cppexp1 = get_variable_name rhs1 vt
                                and cppexp2 = gencpp_scalval rhs2 vt
                                in
                                    "LAMEMatPow(" ^ name ^ ", " ^ cppexp1 ^ ", " ^ cppexp2
    | INIT (_)               ->       "\n"
    | SET  (lhs, rhs1, rhs2, rhs3) -> let name = get_variable_name lhs vt
                                and cppexp1 = gencpp_scalval rhs1 vt
                                and cppexp2 = gencpp_scalval rhs2 vt
                                and cppexp3 = gencpp_scalval rhs3 vt
                                in
                                    "LAMESetElem(" ^ name ^ ", " ^ cppexp1 ^ ", " ^ cppexp
```

```
| GET  (lhs, rhs1, rhs2, rhs3) -> let name = get_variable_name lhs vt
                                  and cppexp1 = gencpp_scalval rhs1 vt
                                  and cppexp2 = gencpp_scalval rhs2 vt
                                  and cppexp3 = get_variable_name rhs3 vt
                                  in
                                      "LAMEGetElem(" ^ cppexp3 ^ ", " ^ name ^ ", " ^ cppexp
| TRAN (lhs, rhs)          -> let name = get_variable_name lhs vt
                                  and cppexp = get_variable_name rhs vt
                                  in
                                      "LAMEMatTrans(" ^ name ^ ", " ^ cppexp ^ ");\n"
| DIM (lhs, rhs1, rhs2)   -> let name = get_variable_name lhs vt
                                  and cppexp1 = gencpp_scalval rhs1 vt
                                  and cppexp2 = gencpp_scalval rhs2 vt
                                  in
                                      "LAMEMatDim(" ^ name ^ ", " ^ cppexp1 ^ ", " ^ cppexp2
| PRINT (lhs)              -> let name = get_variable_name lhs vt
                                  in
                                      "LAMEPrint(" ^ name ^ ");\n"
| SINI (_)                 ->     "\n"
| SCPY (lhs, rhs)          -> let name = get_variable_name lhs vt
                                  and cppexp = get_variable_name rhs vt
                                  in
                                      name ^ " = " ^ cppexp ^ ";\n"
| SFRE (_)                 ->     "\n"
| SCAT (lhs, rhs1, rhs2) -> let name = get_variable_name lhs vt
                                  and cppexp1 = get_variable_name rhs1 vt
                                  and cppexp2 = get_variable_name rhs2 vt
                                  in
                                      "LAMEStrConcat(" ^ name ^ ", " ^ cppexp1 ^ ", " ^ cppe
| SCMP (lhs, rhs1, rhs2) -> let name = get_variable_name lhs vt
                                  and cppexp1 = get_variable_name rhs1 vt
                                  and cppexp2 = get_variable_name rhs2 vt
                                  in
                                      "LAMEStrCmp(" ^ name ^ ", " ^ cppexp1 ^ ", " ^ cppexp2
| SFSC (lhs, rhs)          -> let name = get_variable_name lhs vt
                                  and cppexp = gencpp_scalval rhs vt
                                  in
                                      "LAMEStrFromScalar(" ^ name ^ ", " ^ cppexp ^ ");\n"
| SFMA (lhs, rhs)          -> let name = get_variable_name lhs vt
                                  and cppexp = get_variable_name rhs vt
                                  in
                                      "LAMEStrFromMat(" ^ name ^ ", " ^ cppexp ^ ");\n"
| SLT (lhs, rhs1, rhs2)   -> let name = get_variable_name lhs vt
                                  and cppexp1 = gencpp_scalval rhs1 vt
                                  and cppexp2 = gencpp_scalval rhs2 vt
                                  in
```

```
                                    name ^ " = (" ^ cppexp1 ^ " < " ^ cppexp2 ^ ");\n"
| SLE (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                            and cppexp1 = gencpp_scalval rhs1 vt
                            and cppexp2 = gencpp_scalval rhs2 vt
                            in
                                    name ^ " = (" ^ cppexp1 ^ " <= " ^ cppexp2 ^ ");\n"
| SGT (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                            and cppexp1 = gencpp_scalval rhs1 vt
                            and cppexp2 = gencpp_scalval rhs2 vt
                            in
                                    name ^ " = (" ^ cppexp1 ^ " > " ^ cppexp2 ^ ");\n"
| SGE (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                            and cppexp1 = gencpp_scalval rhs1 vt
                            and cppexp2 = gencpp_scalval rhs2 vt
                            in
                                    name ^ " = (" ^ cppexp1 ^ " >= " ^ cppexp2 ^ ");\n"
| SEQ (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                            and cppexp1 = gencpp_scalval rhs1 vt
                            and cppexp2 = gencpp_scalval rhs2 vt
                            in
                                    name ^ " = (" ^ cppexp1 ^ " == " ^ cppexp2 ^ ");\n"
| SNE (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                            and cppexp1 = gencpp_scalval rhs1 vt
                            and cppexp2 = gencpp_scalval rhs2 vt
                            in
                                    name ^ " = (" ^ cppexp1 ^ " != " ^ cppexp2 ^ ");\n"
| SMEQ (lhs, rhs1, rhs2) -> let name = get_variable_name lhs vt
                            and cppexp1 = get_variable_name rhs1 vt
                            and cppexp2 = get_variable_name rhs2 vt
                            in
                                    name ^ " = (" ^ cppexp1 ^ " == " ^ cppexp2 ^ ");\n"
| SMNE (lhs, rhs1, rhs2) -> let name = get_variable_name lhs vt
                            and cppexp1 = get_variable_name rhs1 vt
                            and cppexp2 = get_variable_name rhs2 vt
                            in
                                    name ^ " = (" ^ cppexp1 ^ " != " ^ cppexp2 ^ ");\n"

| OR (lhs, rhs1, rhs2)   -> let name = get_variable_name lhs vt
                            and cppexp1 = gencpp_boolval rhs1 vt
                            and cppexp2 = gencpp_boolval rhs2 vt
                            in
                                    name ^ " = (" ^ cppexp1 ^ " || " ^ cppexp2 ^ ");\n"
| AND (lhs, rhs1, rhs2)   -> let name = get_variable_name lhs vt
                            and cppexp1 = gencpp_boolval rhs1 vt
                            and cppexp2 = gencpp_boolval rhs2 vt
                            in
```

```
                                    name ^ " = (" ^ cppexp1 ^ " && " ^ cppexp2 ^ ");\n"
      | NOT (lhs, rhs)          -> let name = get_variable_name lhs vt
                                   and cppexp = gencpp_boolval rhs vt
                                   in
                                    name ^ " = !(" ^ cppexp ^ ");\n"
      | BRAF (lhs, rhs)         -> let cppexp = gencpp_boolval lhs vt
                                   and name = get_variable_name rhs vt
                                   in
                                    "if(!(" ^ cppexp ^ ")) goto " ^ name ^ ";\n"
      | JMP (lhs)               -> let name = get_variable_name lhs vt
                                   in
                                    "goto " ^ name ^ ";\n"
      | LABL (lhs)              -> let name = get_variable_name lhs vt
                                   in
                                    name ^ ":\n"
      | ROWS (lhs, rhs)         -> let name = get_variable_name lhs vt
                                   and cppexp = get_variable_name rhs vt
                                   in
                                    "LAMEMatRows(" ^ name ^ ", " ^ cppexp ^ ");\n"
      | COLS (lhs, rhs)         -> let name = get_variable_name lhs vt
                                   and cppexp = get_variable_name rhs vt
                                   in
                                    "LAMEMatCols(" ^ name ^ ", " ^ cppexp ^ ");\n"
      | RDIM (lhs, rows, cols) -> let name = get_variable_name lhs vt
                                   and cppexp1 = gencpp_scalval rows vt
                                   and cppexp2 = gencpp_scalval cols vt
                                   in
                                    "LAMEMatDim(" ^ name ^", " ^ cppexp1 ^ ", " ^ cppexp2

let rec gencpp_prog_rec iprog vt =
    match iprog with
        i :: is -> "\t" ^ (gencpp i vt) ^ (gencpp_prog_rec is vt)
      | []      -> ""

let gencpp_prog iprog vt =
    gencpp_init ^ (gencpp_vartable vt) ^ (gencpp_prog_rec iprog vt) ^ gencpp_end
```

```
type row = float list

type matrix = row list

type lvalue =
    Ident of string
  | VAccess of string * expr
  | MAccess of string * expr * expr

and expr =
    NumLit of float
  | MatLit of matrix
  | StrLit of string
  | BoolLit of bool
  | LValue of lvalue
  | Plus of expr * expr
  | Minus of expr * expr
  | Times of expr * expr
  | Divide of expr * expr
  | Power of expr * expr
  | Eq of expr * expr
  | Neq of expr * expr
  | Lt of expr * expr
  | Gt of expr * expr
  | Le of expr * expr
  | Ge of expr * expr
  | Not of expr
  | And of expr * expr
  | Or of expr * expr
  | Trans of expr
  | SizeRows of expr
  | SizeCols of expr

type datatype =
    Scalar
  | String
  | Matrix
  | Boolean

type stmt =
    Assign of lvalue * expr
  | If of expr * stmt
  | IfElse of expr * stmt * stmt
  | While of expr * stmt
  | Print of expr
  | Dim of string * expr * expr
  | Decl of datatype * string
  | DeclInit of datatype * string * expr
  | StmtList of stmt list

type prgm = stmt list
```

```
(* Muhammad Ali Akbar *)

open Icode
open Vars

let gencpp_init =
    "#include <iostream>\n#include \"lame.h\"\n\nusing namespace std;\n\nint main(void)\n{\n"

let gencpp_end =
    "\treturn 0;\n}\n"

let string_of_bool bv =
    match bv with
        true   -> "TRUE"
      | false  -> "FALSE"

let gencpp_scalval sv vt =
    match sv with
        SLit n -> string_of_float n
      | SVar n -> get_variable_name n vt

let gencpp_boolval bv vt =
    match bv with
        BLit n -> string_of_bool n
      | BVar n -> get_variable_name n vt

let gencpp i vt =
    match i with
        SCAL (lhs, rhs)        -> let name = get_variable_name lhs vt
                                  and cppexp = gencpp_scalval rhs vt
                                  in
                                      name ^ " = " ^ cppexp ^ ";\n"
      | BOOL (lhs, rhs)        -> let name = get_variable_name lhs vt
                                  and cppexp = gencpp_boolval rhs vt
                                  in
                                      name ^ " = " ^ cppexp ^ ";\n"
      | STR (lhs, rhs)         -> let name = get_variable_name lhs vt
                                  and cppexp = rhs
                                  in
                                      name ^ " = " ^ cppexp ^ ";\n"
      | ADD (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                                  and cppexp1 = gencpp_scalval rhs1 vt
                                  and cppexp2 = gencpp_scalval rhs2 vt
                                  in
                                      name ^ " = " ^ cppexp1 ^ " + " ^ cppexp2 ^ ";\n"
      | SUB (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                                  and cppexp1 = gencpp_scalval rhs1 vt
                                  and cppexp2 = gencpp_scalval rhs2 vt
                                  in
                                      name ^ " = " ^ cppexp1 ^ " - " ^ cppexp2 ^ ";\n"
      | MUL (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                                  and cppexp1 = gencpp_scalval rhs1 vt
                                  and cppexp2 = gencpp_scalval rhs2 vt
                                  in
                                      name ^ " = " ^ cppexp1 ^ " * " ^ cppexp2 ^ ";\n"
      | DIV (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                                  and cppexp1 = gencpp_scalval rhs1 vt
                                  and cppexp2 = gencpp_scalval rhs2 vt
                                  in
                                      name ^ " = " ^ cppexp1 ^ " / " ^ cppexp2 ^ ";\n"
      | POW (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                                  and cppexp1 = gencpp_scalval rhs1 vt
                                  and cppexp2 = gencpp_scalval rhs2 vt
                                  in
                                      name ^ " = pow(" ^ cppexp1 ^ ", " ^ cppexp2 ^ ");\n"
      | MADD (lhs, rhs1, rhs2) -> let name = get_variable_name lhs vt
```

```
                                and cppexp1 = get_variable_name rhs1 vt
                                and cppexp2 = get_variable_name rhs2 vt
                                in
                                    name ^ " = " ^ cppexp1 ^ " + " ^ cppexp2 ^ ";\n"
    | MSUB (lhs, rhs1, rhs2) -> let name = get_variable_name lhs vt
                                and cppexp1 = get_variable_name rhs1 vt
                                and cppexp2 = get_variable_name rhs2 vt
                                in
                                    name ^ " = " ^ cppexp1 ^ " - " ^ cppexp2 ^ ";\n"
    | MMUL (lhs, rhs1, rhs2) -> let name = get_variable_name lhs vt
                                and cppexp1 = get_variable_name rhs1 vt
                                and cppexp2 = get_variable_name rhs2 vt
                                in
                                    name ^ " = " ^ cppexp1 ^ " * " ^ cppexp2 ^ ";\n"
    | SMUL (lhs, rhs1, rhs2) -> let name = get_variable_name lhs vt
                                and cppexp1 = get_variable_name rhs1 vt
                                and cppexp2 = gencpp_scalval rhs2 vt
                                in
                                    "LAMEScalarMul(" ^ name ^ ", " ^ cppexp1 ^ ", " ^ cppexp2 ^ ");\n"
    | MPOW (lhs, rhs1, rhs2) -> let name = get_variable_name lhs vt
                                and cppexp1 = get_variable_name rhs1 vt
                                and cppexp2 = gencpp_scalval rhs2 vt
                                in
                                    "LAMEMatPow(" ^ name ^ ", " ^ cppexp1 ^ ", " ^ cppexp2 ^ ");\n"
    | INIT (_)               ->       "\n"
    | SET  (lhs, rhs1, rhs2, rhs3) -> let name = get_variable_name lhs vt
                                and cppexp1 = gencpp_scalval rhs1 vt
                                and cppexp2 = gencpp_scalval rhs2 vt
                                and cppexp3 = gencpp_scalval rhs3 vt
                                in
                                    "LAMESetElem(" ^ name ^ ", " ^ cppexp1 ^ ", " ^ cppexp2 ^ ", " ^      ↙
cppexp3 ^ ");\n"
    | GET  (lhs, rhs1, rhs2, rhs3) -> let name = get_variable_name lhs vt
                                and cppexp1 = gencpp_scalval rhs1 vt
                                and cppexp2 = gencpp_scalval rhs2 vt
                                and cppexp3 = get_variable_name rhs3 vt
                                in
                                    "LAMEGetElem(" ^ cppexp3 ^ ", " ^ name ^ ", " ^ cppexp1 ^ ", " ^      ↙
cppexp2 ^ ");\n"
    | TRAN (lhs, rhs)        -> let name = get_variable_name lhs vt
                                and cppexp = get_variable_name rhs vt
                                in
                                    "LAMEMatTrans(" ^ name ^ ", " ^ cppexp ^ ");\n"
    | DIM (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                                and cppexp1 = gencpp_scalval rhs1 vt
                                and cppexp2 = gencpp_scalval rhs2 vt
                                in
                                    "LAMEMatDim(" ^ name ^ ", " ^ cppexp1 ^ ", " ^ cppexp2 ^ ");\n"
    | PRINT (lhs)            -> let name = get_variable_name lhs vt
                                in
                                    "LAMEPrint(" ^ name ^ ");\n"
    | SINI (_)               ->     "\n"
    | SCPY (lhs, rhs)        -> let name = get_variable_name lhs vt
                                and cppexp = get_variable_name rhs vt
                                in
                                    name ^ " = " ^ cppexp ^ ";\n"
    | SFRE (_)               ->     "\n"
    | SCAT (lhs, rhs1, rhs2) -> let name = get_variable_name lhs vt
                                and cppexp1 = get_variable_name rhs1 vt
                                and cppexp2 = get_variable_name rhs2 vt
                                in
                                    "LAMEStrConcat(" ^ name ^ ", " ^ cppexp1 ^ ", " ^ cppexp2 ^ ");\n"
    | SCMP (lhs, rhs1, rhs2) -> let name = get_variable_name lhs vt
                                and cppexp1 = get_variable_name rhs1 vt
                                and cppexp2 = get_variable_name rhs2 vt
                                in
```

```
                                    "LAMEStrCmp(" ^ name ^ ", " ^ cppexp1 ^ ", " ^ cppexp2 ^ ");\n"
      | SFSC (lhs, rhs)        -> let name = get_variable_name lhs vt
                                  and cppexp = gencpp_scalval rhs vt
                                  in
                                    "LAMEStrFromScalar(" ^ name ^ ", " ^ cppexp ^ ");\n"
      | SFMA (lhs, rhs)        -> let name = get_variable_name lhs vt
                                  and cppexp = get_variable_name rhs vt
                                  in
                                    "LAMEStrFromMat(" ^ name ^ ", " ^ cppexp ^ ");\n"
      | SLT (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                                  and cppexp1 = gencpp_scalval rhs1 vt
                                  and cppexp2 = gencpp_scalval rhs2 vt
                                  in
                                    name ^ " = (" ^ cppexp1 ^ " < " ^ cppexp2 ^ ");\n"
      | SLE (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                                  and cppexp1 = gencpp_scalval rhs1 vt
                                  and cppexp2 = gencpp_scalval rhs2 vt
                                  in
                                    name ^ " = (" ^ cppexp1 ^ " <= " ^ cppexp2 ^ ");\n"
      | SGT (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                                  and cppexp1 = gencpp_scalval rhs1 vt
                                  and cppexp2 = gencpp_scalval rhs2 vt
                                  in
                                    name ^ " = (" ^ cppexp1 ^ " > " ^ cppexp2 ^ ");\n"
      | SGE (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                                  and cppexp1 = gencpp_scalval rhs1 vt
                                  and cppexp2 = gencpp_scalval rhs2 vt
                                  in
                                    name ^ " = (" ^ cppexp1 ^ " >= " ^ cppexp2 ^ ");\n"
      | SEQ (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                                  and cppexp1 = gencpp_scalval rhs1 vt
                                  and cppexp2 = gencpp_scalval rhs2 vt
                                  in
                                    name ^ " = (" ^ cppexp1 ^ " == " ^ cppexp2 ^ ");\n"
      | SNE (lhs, rhs1, rhs2)  -> let name = get_variable_name lhs vt
                                  and cppexp1 = gencpp_scalval rhs1 vt
                                  and cppexp2 = gencpp_scalval rhs2 vt
                                  in
                                    name ^ " = (" ^ cppexp1 ^ " != " ^ cppexp2 ^ ");\n"
      | SMEQ (lhs, rhs1, rhs2) -> let name = get_variable_name lhs vt
                                  and cppexp1 = get_variable_name rhs1 vt
                                  and cppexp2 = get_variable_name rhs2 vt
                                  in
                                    name ^ " = (" ^ cppexp1 ^ " == " ^ cppexp2 ^ ");\n"
      | SMNE (lhs, rhs1, rhs2) -> let name = get_variable_name lhs vt
                                  and cppexp1 = get_variable_name rhs1 vt
                                  and cppexp2 = get_variable_name rhs2 vt
                                  in
                                    name ^ " = (" ^ cppexp1 ^ " != " ^ cppexp2 ^ ");\n"

      | OR (lhs, rhs1, rhs2)   -> let name = get_variable_name lhs vt
                                  and cppexp1 = gencpp_boolval rhs1 vt
                                  and cppexp2 = gencpp_boolval rhs2 vt
                                  in
                                    name ^ " = (" ^ cppexp1 ^ " || " ^ cppexp2 ^ ");\n"
      | AND (lhs, rhs1, rhs2)   -> let name = get_variable_name lhs vt
                                  and cppexp1 = gencpp_boolval rhs1 vt
                                  and cppexp2 = gencpp_boolval rhs2 vt
                                  in
                                    name ^ " = (" ^ cppexp1 ^ " && " ^ cppexp2 ^ ");\n"
      | NOT (lhs, rhs)         -> let name = get_variable_name lhs vt
                                  and cppexp = gencpp_boolval rhs vt
                                  in
                                    name ^ " = !(" ^ cppexp ^ ");\n"
      | BRAF (lhs, rhs)        -> let cppexp = gencpp_boolval lhs vt
                                  and name = get_variable_name rhs vt
```

```
                                in
                                    "if(!(" ^ cppexp ^ ")) goto " ^ name ^ ";\n"
      | JMP (lhs)             -> let name = get_variable_name lhs vt
                                in
                                    "goto " ^ name ^ ";\n"
      | LABL (lhs)            -> let name = get_variable_name lhs vt
                                in
                                    name ^ ":\n"
      | ROWS (lhs, rhs)       -> let name = get_variable_name lhs vt
                                and cppexp = get_variable_name rhs vt
                                in
                                    "LAMEMatRows(" ^ name ^ ", " ^ cppexp ^ ");\n"
      | COLS (lhs, rhs)       -> let name = get_variable_name lhs vt
                                and cppexp = get_variable_name rhs vt
                                in
                                    "LAMEMatCols(" ^ name ^ ", " ^ cppexp ^ ");\n"
      | RDIM (lhs, rows, cols) -> let name = get_variable_name lhs vt
                                and cppexp1 = gencpp_scalval rows vt
                                and cppexp2 = gencpp_scalval cols vt
                                in
                                    "LAMEMatDim(" ^ name ^", " ^ cppexp1 ^ ", " ^ cppexp2 ^ ");\n"

let rec gencpp_prog_rec iprog vt =
    match iprog with
        i :: is -> "\t" ^ (gencpp i vt) ^ (gencpp_prog_rec is vt)
      | []      -> ""

let gencpp_prog iprog vt =
    gencpp_init ^ (gencpp_vartable vt) ^ (gencpp_prog_rec iprog vt) ^ gencpp_end
```

```
(* Muhammad Ali Akbar *)

open Icode
open Vars

val gencpp_prog : iprog -> vartable -> string
```

```
type scalval =
    SVar of int
  | SLit of float

type boolval =
    BVar of int
  | BLit of bool

type intinst =
    SCAL of int * scalval
  | BOOL of int * boolval
  | STR of int * string
  | ADD of int * scalval * scalval
  | SUB of int * scalval * scalval
  | MUL of int * scalval * scalval
  | DIV of int * scalval * scalval
  | POW of int * scalval * scalval
  | MADD of int * int * int
  | MSUB of int * int * int
  | MMUL of int * int * int
  | SMUL of int * int * scalval
  | MPOW of int * int * scalval
  | INIT of int
  | SET of int * scalval * scalval * scalval
  | GET of int * scalval * scalval * int
  | TRAN of int * int
  | DIM of int * scalval * scalval
  | PRINT of int
  | SINI of int
  | SCPY of int * int
  | SFRE of int
  | SCAT of int * int * int
  | SCMP of int * int * int
  | SFSC of int * scalval
  | SFMA of int * int
  | SLT of int * scalval * scalval
  | SLE of int * scalval * scalval
  | SGT of int * scalval * scalval
  | SGE of int * scalval * scalval
  | SEQ of int * scalval * scalval
  | SNE of int * scalval * scalval
  | SMEQ of int * int * int
  | SMNE of int * int * int
  | OR of int * boolval * boolval
  | AND of int * boolval * boolval
  | NOT of int * boolval
  | BRAF of boolval * int
  | JMP of int
  | LABL of int
  | ROWS of int * int
  | COLS of int * int
  | RDIM of int * scalval * scalval

type iprog = intinst list
```

```
(* David Golub *)

type labels = { mutable next : int }

let new_labels () = { next = 0 }
let add_label ls =
    let labno = ls.next
    in ls.next <- labno + 1;
       labno
let get_label_name n =
    "_label" ^ string_of_int n
```

```
(* David Golub *)

open Ast

type labels

val new_labels : unit -> labels
val add_label : labels -> int
val get_label_name : int -> string
```

```
/* Muhammad Ali Akbar */

#ifndef LAME_H
#define LAME_H

#include "matrix.h"

inline void LAMEStrFromScalar(String &s, Scalar x)
{
    ostringstream strs;
    strs<<x;
    s = strs.str();
}

inline void LAMEStrFromMat(String &s, Matrix &m)
{
    s = m.toString();
}

inline void LAMEMatAdd(Matrix &m, Matrix &m1, Matrix &m2)
{
    m = m1 + m2;
}

inline void LAMEMatSub(Matrix &m, Matrix &m1, Matrix &m2)
{
    m = m1 - m2;
}

inline void LAMEMatMul(Matrix &m, Matrix &m1, Matrix &m2)
{
    m = m1 * m2;
}

inline void LAMEScalarMul(Matrix &m, Matrix &m1, Scalar s2)
{
    m = m1 * s2;
}

inline void LAMEMatPow(Matrix &m, Matrix &m1, Scalar s2)
{
    m = m1.power(s2);
}

inline void LAMESetElem(Matrix &m, unsigned int i, unsigned int j, Scalar s)
{
    m.setElement(i,j,s);
}

inline void LAMEGetElem(Scalar &s, Matrix &m, unsigned int i, unsigned int j)
{
    s=m.getElement(i,j);
}

inline void LAMEMatTrans(Matrix &m1, Matrix &m2)
{
    m1=m2.transpose();
}

inline void LAMEMatDim(Matrix &m, unsigned int i, unsigned int j)
{
    m.reDim(i,j);
}

inline void LAMEMatRows(Scalar &s, Matrix &m)
{
```

```
    s=m.getNumRows();
}

inline void LAMEMatCols(Scalar &s, Matrix &m)
{
    s=m.getNumCols();
}

inline void LAMEMatEqual(Boolean b, Matrix &m1, Matrix &m2)
{
    b=(m1==m2);
}

inline void LAMEMatNotEqual(Boolean b, Matrix &m1, Matrix &m2)
{
    b=!(m1==m2);
}

inline void LAMEPrint(String s)
{
    cout<<s;
}

inline void LAMEStrCopy(String &s1, String s2)
{
    s1 = s2;
}

inline void LAMEStrConcat(String &s, String &s1, String &s2)
{
    s = s1 + s2;
}

inline void LAMEStrCmp(Boolean &b, String &s1, String &s2)
{
    b = (s1==s2);
}

inline void LAMEStrFree(String &s)
{
}

inline void LAMEStrInit(String &s)
{
    s = "";
}

inline void LAMEMatInit(Matrix &m)
{
}

#endif //LAME_H
```

```
open Gencpp
open Labels
open Semantic
open Vars

let _ =
let lexbuf = Lexing.from_channel stdin
in let program = Lamepar.prgm Lamescan.token lexbuf
in let vt = new_vartable ()
in let ls = new_labels ()
in let intcode = check_prgm program vt ls
in let cppcode = gencpp_prog intcode vt
in print_endline cppcode
```

```
%{
    (* Carmine Elvezio *)
    open Ast
%}

%token PLUS MINUS TIMES DIVIDE POWER COMMA SEMICOLON
%token LBRACE RBRACE LPAREN RPAREN LBRACK RBRACK
%token ASSIGN EQ NE LT LE GT GE NOT OR AND TRANS
%token IF ELSE WHILE PRINT TRUE FALSE
%token STRING SCALAR MATRIX BOOLEAN
%token SIZE_ROWS SIZE_COLS DIM
%token EOF
%token <string> IDENT
%token <string> STRLIT
%token <float> NUM

%nonassoc NOELSE
%nonassoc ELSE

%nonassoc NOSEMI
%nonassoc SEMICOLON

%nonassoc NOCOMMA
%nonassoc COMMA

%left OR
%left AND
%nonassoc EQ NE
%left LT LE GT GE
%left PLUS MINUS
%left TIMES DIVIDE
%left POWER
%nonassoc NOT
%nonassoc UMINUS
%nonassoc TRANS
%nonassoc SIZE_ROWS SIZE_COLS

%start prgm
%type <float list> row
%type <row list> matrix
%type <lvalue> lvalue
%type <expr> expr
%type <datatype> datatype
%type <stmt> stmt
%type <stmt list> stmtlist
%type <Ast.stmt list> prgm

%%

row:
  NUM COMMA row         {$1 :: $3}
| NUM %prec NOCOMMA     {[$1]}

matrix:
  row SEMICOLON matrix  {$1 :: $3}
| row %prec NOSEMI      {[$1]}

lvalue:
  IDENT                                     {Ident($1)}
| IDENT LBRACK expr RBRACK                  {VAccess($1, $3)}
| IDENT LBRACK expr COMMA expr RBRACK       {MAccess($1, $3, $5)}

expr:
  NUM                       {NumLit($1)}
| STRLIT                     {StrLit($1)}
| TRUE                       {BoolLit(true)}
```

```
| FALSE                     {BoolLit(false)}
| lvalue                    {LValue($1)}
| expr PLUS expr            {Plus($1, $3)}
| expr MINUS expr           {Minus($1, $3)}
| expr TIMES expr           {Times($1, $3)}
| expr DIVIDE expr          {Divide($1, $3)}
| expr POWER expr           {Power($1, $3)}
| MINUS expr %prec UMINUS   {Times(NumLit(-1.0), $2)}
| expr EQ expr              {Eq($1, $3)}
| expr NE expr              {Neq($1, $3)}
| expr LT expr              {Lt($1, $3)}
| expr GT expr              {Gt($1, $3)}
| expr LE expr              {Le($1, $3)}
| expr GE expr              {Ge($1, $3)}
| NOT expr                  {Not($2)}
| expr AND expr             {And($1, $3)}
| expr OR expr              {Or($1, $3)}
| expr TRANS                {Trans($1)}
| LPAREN expr RPAREN        {$2}
| LBRACE matrix RBRACE      {MatLit($2)}
| SIZE_ROWS expr            {SizeRows($2)}
| SIZE_COLS expr            {SizeCols($2)}

datatype:
  SCALAR               {Scalar}
| STRING               {String}
| MATRIX               {Matrix}
| BOOLEAN              {Boolean}

stmt:
  lvalue ASSIGN expr SEMICOLON                     {Assign($1, $3)}
| IF LPAREN expr RPAREN stmt %prec NOELSE           {If($3, $5) }
| IF LPAREN expr RPAREN stmt ELSE stmt              {IfElse($3, $5, $7)}
| WHILE LPAREN expr RPAREN stmt                     {While($3, $5)}
| PRINT expr SEMICOLON                              {Print($2)}
| DIM IDENT LBRACK expr COMMA expr RBRACK SEMICOLON {Dim($2, $4, $6)}
| datatype IDENT SEMICOLON                          {Decl($1, $2)}
| datatype IDENT ASSIGN expr SEMICOLON              {DeclInit($1, $2, $4)}
| LBRACE stmtlist RBRACE                            {StmtList($2)}

stmtlist:
  stmt stmtlist                          {$1 :: $2}
| stmt                                   {[$1]}

prgm:
  stmt EOF             {[$1]}
| stmt prgm            {$1 :: $2}
```

```
{ open Lamepar }

rule token =
  parse [' ' '\t' '\r' '\n'] { token lexbuf }
      | eof                  { EOF }
      | '+'                  { PLUS }
      | '-'                  { MINUS }
      | '*'                  { TIMES }
      | '/'                  { DIVIDE }
      | '^'                  { POWER }
      | ','                  { COMMA }
      | ';'                  { SEMICOLON }
      | '{'                  { LBRACE }
      | '}'                  { RBRACE }
      | '('                  { LPAREN }
      | ')'                  { RPAREN }
      | '['                  { LBRACK }
      | ']'                  { RBRACK }
      | "=="                 { EQ }
      | "!="                 { NE }
      | '<'                  { LT }
      | "<="                 { LE }
      | '>'                  { GT }
      | ">="                 { GE }
      | '!'                  { NOT }
      | "||"                 { OR }
      | "&&"                 { AND }
      | '\''                 { TRANS }
      | '='                  { ASSIGN }
      | "dim"                { DIM }
      | "if"                 { IF }
      | "else"               { ELSE }
      | "while"              { WHILE }
      | "print"              { PRINT }
      | "true"               { TRUE }
      | "false"              { FALSE }
      | "scalar"             { SCALAR }
      | "string"             { STRING }
      | "matrix"             { MATRIX }
      | "boolean"            { BOOLEAN }
      | "size_cols"          { SIZE_COLS }
      | "size_rows"          { SIZE_ROWS }
      | ['A'-'Z' 'a'-'z' '_']['A'-'Z' 'a'-'z' '0'-'9' '_']* as id { IDENT(id) }
      | ['0'-'9']+ ('.' ['0'-'9']+)? (['e' 'E'] ['-' '+']? ['0'-'9']+)? as num { NUM(float_of_string num) }
      | '"' ([^ '"' '\n' '\r' '\\'] | "\\\\" | "\\\"" | "\\n" | "\\t")* '"' as str { STRLIT(str) }
```

```
.SUFFIXES : .ml .cmo .mli .cmi

CMOS = \
    labels.cmo \
    vars.cmo \
    gencpp.cmo \
    semantic.cmo \
    lamescan.cmo \
    lamepar.cmo \
    lame.cmo

CMIS = \
    ast.cmi \
    labels.cmi \
    vars.cmi \
    icode.cmi \
    gencpp.cmi \
    semantic.cmi \
    lamepar.cmi

all : lame.exe

.ml.cmo :
    ocamlc -c $<

.mli.cmi :
    ocamlc -c $<

lamepar.mli : lamepar.mly
    ocamlyacc lamepar.mly

lamescan.ml : lamescan.mll
    ocamllex lamescan.mll

lame.exe : $(CMIS) $(CMOS)
    ocamlc -o lame.exe $(CMOS)
```

```
CMOS = labels.cmo vars.cmo gencpp.cmo semantic.cmo lamescan.cmo lamepar.cmo lame.cmo

CMIS = ast.cmi labels.cmi vars.cmi icode.cmi gencpp.cmi semantic.cmi lamepar.cmi

lame : $(CMIS) $(CMOS)
    ocamlc -o lame $(CMOS)

%.cmo : %.ml
    ocamlc -c $<

%.cmi : %.mli
    ocamlc -c $<

lamepar.mli : lamepar.mly
    ocamlyacc lamepar.mly

lamescan.ml : lamescan.mll
    ocamllex lamescan.mll

lamepar.cmo : lamepar.ml
```

```
/* Muhammad Ali Akbar */

#ifndef MATRIX_H
#define MATRIX_H

#include <iostream>
#include <vector>
#include <cmath>
#include <cstring>
#include <sstream>
#include <assert.h>
using namespace std;

typedef double Scalar;
typedef bool Boolean;
typedef string String;

class Matrix
{
    vector< vector <double> > v;
public:
    Matrix(unsigned int r=1, unsigned int c=1)
    {
        assert(r>0);
        assert(c>0);

        for(unsigned int i=0; i<r; i++)
        {
            v.push_back(vector<double>());
            for(unsigned int j=0; j<c; j++)
            {
                v[i].push_back(0);
            }
        }
    }
    Matrix(const Matrix &x)
    {
        v.clear();
        for(unsigned int i=0; i<x.v.size(); i++)
        {
            v.push_back(vector<double>());
            for(unsigned int j=0; j<x.v[i].size(); j++)
            {
                v[i].push_back(x.v[i][j]);
            }
        }
    }
    void reDim(unsigned int i, unsigned int j)
    {
        unsigned int nRows = getNumRows();
        unsigned int nCols = getNumCols();

        for(unsigned int indi=i; indi<nRows; indi++)
        {
            v.pop_back();
        }

        for(unsigned int indj=j; indj<nCols; indj++)
        {
            for(unsigned int indi=0; indi<i; indi++)
            {
                v[indi].pop_back();
            }
        }

        nRows = getNumRows();
```

```
        nCols = getNumCols();

        if(i<nRows && j<nCols)
        {
            return;
        }

        for(unsigned int indi=0; indi<i; indi++)
        {
            if(indi>=nRows)
            {
                v.push_back(vector<double>());
            }
            for(unsigned int indj=0; indj<j; indj++)
            {
                if(indi>=nRows)
                {
                    v[indi].push_back(0);
                }
                else if(indj>=nCols)
                {
                    v[indi].push_back(0);
                }
            }
        }
    }
    Matrix transpose()
    {
        unsigned int nRows = getNumRows();
        unsigned int nCols = getNumCols();

        Matrix m(nCols,nRows);

        for(unsigned int i=0; i<m.v.size(); i++)
        {
            for(unsigned int j=0; j<m.v[i].size(); j++)
            {
                m.v[i][j]=v[j][i];
            }
        }
        return m;
    }
    Matrix power(unsigned int x)
    {
        if(getNumRows() != getNumCols()) {
            printf("Runtime error: Matrix power requires a square matrix!\n");
            exit(0);
        }

        Matrix m(getNumRows(), getNumCols());
        for(unsigned int i=0; i<getNumRows(); i++)
        {
            m.setElement(i, i, 1.0);
        }
        for(unsigned int i=1; i<=x; i++)
        {
            m = m * *this;
        }
        return m;
    }
    void setElement(unsigned int i, unsigned int j, Scalar x)
    {
        //assert(i<v.size());
        //assert(j<v[i].size());

        if(i>=v.size())
```

```
        {
            reDim(i+1,getNumCols());
        }

        if(j>=v[i].size())
        {
            reDim(getNumRows(),j+1);
        }

        v[i][j]=x;
    }
    Scalar getElement(unsigned int i, unsigned int j)
    {
        if(i>=v.size() || j>=v[i].size())
        {
            printf("Runtime error: Index out of bounds!\n");
            exit(0);
        }

        return v[i][j];
    }
    unsigned int getNumRows()
    {
        return v.size();
    }
    String toString()
    {
        ostringstream strs;
        for(unsigned int i=0; i<v.size(); i++)
        {
            for(unsigned int j=0; j<v[i].size(); j++)
            {
                strs<<v[i][j]<<"\t";
            }
            strs<<"\n";
        }
        return strs.str();
    }
    unsigned int getNumCols()
    {
        return v[0].size();
    }
    bool operator == (Matrix& m)
    {
        unsigned int nRows = getNumRows();
        unsigned int nCols = getNumCols();

        assert(nRows==m.getNumRows());
        assert(nCols==m.getNumCols());

        bool isEqual = true;

        for(unsigned int i=0; i<m.v.size(); i++)
        {
            for(unsigned int j=0; j<m.v[i].size(); j++)
            {
                if(v[i][j]!=m.v[i][j])
                {
                    isEqual = false;
                    break;
                }
            }
            if(!isEqual)
            {
                break;
            }
```

```
        }
        return isEqual;
    }
    const Matrix& operator = (const Matrix &x)
    {
        v.clear();
        for(unsigned int i=0; i<x.v.size(); i++)
        {
            v.push_back(vector<double>());
            for(unsigned int j=0; j<x.v[i].size(); j++)
            {
                v[i].push_back(x.v[i][j]);
            }
        }
        return *this;
    }
    friend ostream& operator << (ostream& os, const Matrix& m);
    friend Matrix operator + (Matrix& m1, Matrix& m2);
    friend Matrix operator - (Matrix& m1, Matrix& m2);
    friend Matrix operator * (Matrix& m1, Matrix& m2);
    friend Matrix operator * (Matrix& m1, Scalar& s2);
    friend Matrix operator - (Matrix& m1);
};

ostream& operator << (ostream& os, const Matrix& m)
{
    for(unsigned int i=0; i<m.v.size(); i++)
    {
        for(unsigned int j=0; j<m.v[i].size(); j++)
        {
            os<<m.v[i][j]<<"\t";
        }
        os<<"\n";
    }
    return os;
}

Matrix operator + (Matrix& m1, Matrix& m2)
{
    unsigned int nRows = m1.getNumRows();
    unsigned int nCols = m1.getNumCols();

    assert(nRows==m2.getNumRows());
    assert(nCols==m2.getNumCols());

    Matrix m(nRows,nCols);
    for(unsigned int i=0; i<m.v.size(); i++)
    {
        for(unsigned int j=0; j<m.v[i].size(); j++)
        {
            m.v[i][j]=m1.v[i][j]+m2.v[i][j];
        }
    }
    return m;
}

Matrix operator - (Matrix& m1, Matrix& m2)
{
    unsigned int nRows = m1.getNumRows();
    unsigned int nCols = m1.getNumCols();

    assert(nRows==m2.getNumRows());
    assert(nCols==m2.getNumCols());

    Matrix m(nRows,nCols);
    for(unsigned int i=0; i<m.v.size(); i++)
```

```
(* David Golub *)

open Ast
open Icode
open Labels
open Vars

exception InvalidMatrix
exception TypeMismatch
exception WrongDataType

let rec check_cols row temp i j =
    match row with
        hd :: tl -> let sets = check_cols tl temp i (j + 1)
                    in
                        SET (temp, SLit (float_of_int i),
                             SLit (float_of_int j), SLit hd) :: sets
      | []       -> []

let rec check_rows m temp i =
    match m with
        hd :: tl -> let cols1 = List.length hd
                    and (cols2, sets2) = check_rows tl temp (i + 1)
                    in
                        if cols1 = cols2 || cols2 = 0 then
                            let sets1 = check_cols hd temp i 0
                            in
                                (cols1, (sets1 @ sets2))
                        else
                            raise InvalidMatrix
      | []       -> (0, [])

let check_matrix m vt =
    let temp = add_temp Matrix vt
    in
        let (_, sets) = check_rows m temp 0
        in
            let instrs = [INIT temp] @ sets
            in (temp, instrs)

let rec check_expr e vt =
    match e with
        NumLit n        -> let temp = add_temp Scalar vt
                           in
                               let instrs = [SCAL (temp, SLit n)]
                               in (Scalar, temp, instrs)
      | MatLit m        -> let (temp, instrs) = check_matrix m vt
                           in (Matrix, temp, instrs)
      | StrLit s        -> let temp = add_temp String vt
                           in
                               let instrs = [SINI temp;
                                             STR (temp, s)]
                               in (String, temp, instrs)
      | BoolLit b       -> let temp = add_temp Boolean vt
                           in
                               let instrs = [BOOL (temp, BLit b)]
                               in (Boolean, temp, instrs)
      | LValue lv       -> check_lvalue lv vt
      | Plus (e1, e2)   -> let (typ1, temp1, instrs1) = check_expr e1 vt
                           and (typ2, temp2, instrs2) = check_expr e2 vt
                           in
                               if typ1 = Scalar && typ2 = Scalar then
                                   let temp = add_temp Scalar vt
                                   in
                                       let instrs = instrs1 @ instrs2 @
                                                    [ADD (temp, SVar temp1, SVar temp2)]
```

```
                                in (Scalar, temp, instrs)
                            else if typ1 = String && typ2 = String then
                                let temp = add_temp String vt
                                in
                                    let instrs = instrs1 @ instrs2 @
                                                 [SCAT (temp, temp1, temp2)]
                                    in (String, temp, instrs)
                            else if typ1 = Matrix && typ2 = Matrix then
                                let temp = add_temp Matrix vt
                                in
                                    let instrs = instrs1 @ instrs2 @
                                                 [INIT temp;
                                                  MADD (temp, temp1, temp2)]
                                    in (Matrix, temp, instrs)
                            else if typ1 = Boolean && typ2 = Boolean then
                                raise WrongDataType
                            else if typ1 = String && typ2 = Scalar then
                                let temp2prime = add_temp String vt
                                and temp = add_temp String vt
                                in
                                    let instrs = instrs1 @ instrs2 @
                                                 [SINI temp2prime;
                                                  SFSC (temp2prime, SVar temp2);
                                                  SINI temp;
                                                  SCAT (temp, temp1, temp2prime)]
                                    in (String, temp, instrs)
                            else if typ1 = Scalar && typ2 = String then
                                let temp1prime = add_temp String vt
                                and temp = add_temp String vt
                                in
                                    let instrs = instrs1 @ instrs2 @
                                                 [SINI temp1prime;
                                                  SFSC (temp1prime, SVar temp1);
                                                  SINI temp;
                                                  SCAT (temp, temp1prime, temp2)]
                                    in (String, temp, instrs)
                            else if typ1 = String && typ2 = Matrix then
                                let temp2prime = add_temp String vt
                                and temp = add_temp String vt
                                in
                                    let instrs = instrs1 @ instrs2 @
                                                 [SINI temp2prime;
                                                  SFMA (temp2prime, temp2);
                                                  SINI temp;
                                                  SCAT (temp, temp1, temp2prime)]
                                    in (String, temp, instrs)
                            else if typ1 = Matrix && typ2 = String then
                                let temp1prime = add_temp String vt
                                and temp = add_temp String vt
                                in
                                    let instrs = instrs1 @ instrs2 @
                                                 [SINI temp1prime;
                                                  SFMA (temp1prime, temp1);
                                                  SINI temp;
                                                  SCAT (temp, temp1prime, temp2)]
                                    in (String, temp, instrs)
                            else
                                raise TypeMismatch
    | Minus (e1, e2)  -> let (typ1, temp1, instrs1) = check_expr e1 vt
                         and (typ2, temp2, instrs2) = check_expr e2 vt
                         in
                            if typ1 <> typ2 then
                                raise TypeMismatch
                            else if typ1 = Scalar then
                                let temp = add_temp Scalar vt
                                in
```

```
                                let instrs = instrs1 @ instrs2 @
                                             [SUB (temp, SVar temp1, SVar temp2)]
                                in (Scalar, temp, instrs)
                            else if typ1 = Matrix then
                                let temp = add_temp Matrix vt
                                in
                                    let instrs = instrs1 @ instrs2 @
                                                 [INIT temp;
                                                  MSUB (temp, temp1, temp2)]
                                    in (Matrix, temp, instrs)
                            else
                                raise WrongDataType
      | Times (e1, e2)  -> let (typ1, temp1, instrs1) = check_expr e1 vt
                           and (typ2, temp2, instrs2) = check_expr e2 vt
                           in
                               if typ1 = Scalar && typ2 = Scalar then
                                   let temp = add_temp Scalar vt
                                   in
                                       let instrs = instrs1 @ instrs2 @
                                                    [MUL (temp, SVar temp1, SVar temp2)]
                                       in (Scalar, temp, instrs)
                               else if typ1 = Matrix && typ2 = Matrix then
                                   let temp = add_temp Matrix vt
                                   in
                                       let instrs = instrs1 @ instrs2 @
                                                    [INIT temp;
                                                     MMUL (temp, temp1, temp2)]
                                       in (Matrix, temp, instrs)
                               else if typ1 = Scalar && typ2 = Matrix then
                                   let temp = add_temp Matrix vt
                                   in
                                       let instrs = instrs1 @ instrs2 @
                                                    [INIT temp;
                                                     SMUL (temp, temp2, SVar temp1)]
                                       in (Matrix, temp, instrs)
                               else if typ1 = Matrix && typ2 = Scalar then
                                   let temp = add_temp Matrix vt
                                   in
                                       let instrs = instrs1 @ instrs2 @
                                                    [INIT temp;
                                                     SMUL (temp, temp1, SVar temp2)]
                                       in (Matrix, temp, instrs)
                               else
                                   raise WrongDataType
      | Divide (e1, e2) -> let (typ1, temp1, instrs1) = check_expr e1 vt
                           and (typ2, temp2, instrs2) = check_expr e2 vt
                           in
                               if typ1 = Scalar && typ2 = Scalar then
                                   let temp = add_temp Scalar vt
                                   in
                                       let instrs = instrs1 @ instrs2 @
                                                    [DIV (temp, SVar temp1, SVar temp2)]
                                       in (Scalar, temp, instrs)
                               else if typ1 = Matrix && typ2 = Scalar then
                                   let temp = add_temp Matrix vt
                                   and temp2prime = add_temp Scalar vt
                                   in
                                       let instrs = instrs1 @ instrs2 @
                                                    [DIV (temp2prime, SLit 1.0, SVar temp2);
                                                     INIT temp;
                                                     SMUL (temp, temp1, SVar temp2prime)]
                                       in (Matrix, temp, instrs)
                               else
                                   raise WrongDataType
      | Power (e1, e2)  -> let (typ1, temp1, instrs1) = check_expr e1 vt
                           and (typ2, temp2, instrs2) = check_expr e2 vt
```

```
                        in
                            if typ1 = Scalar && typ2 = Scalar then
                                let temp = add_temp Scalar vt
                                in
                                    let instrs = instrs1 @ instrs2 @
                                                 [POW (temp, SVar temp1, SVar temp2)]
                                    in (Scalar, temp, instrs)
                            else if typ1 = Matrix && typ2 = Scalar then
                                let temp = add_temp Matrix vt
                                in
                                    let instrs = instrs1 @ instrs2 @
                                                 [INIT temp;
                                                  MPOW (temp, temp1, SVar temp2)]
                                    in (Matrix, temp, instrs)
                            else
                                raise WrongDataType
    | Eq (e1, e2)     -> let (typ1, temp1, instrs1) = check_expr e1 vt
                        and (typ2, temp2, instrs2) = check_expr e2 vt
                        in
                            if typ1 = Matrix && typ2 = Matrix then
                                let temp = add_temp Boolean vt
                                in
                                    let instrs = instrs1 @ instrs2 @
                                                 [SMEQ (temp, temp1, temp2)]
                                    in (Boolean, temp, instrs)
                            else if typ1 = String && typ2 = String then
                                let cmp_temp = add_temp Scalar vt
                                and temp = add_temp Boolean vt
                                in
                                    let instrs = instrs1 @ instrs2 @
                                                 [SCMP (cmp_temp, temp1, temp2);
                                                  SEQ (temp, SVar cmp_temp, SLit 0.0)]
                                    in (Boolean, temp, instrs)
                            else if typ1 = typ2 then
                                let temp = add_temp Boolean vt
                                in
                                    let instrs = instrs1 @ instrs2 @
                                                 [SEQ (temp, SVar temp1, SVar temp2)]
                                    in (Boolean, temp, instrs)
                            else
                                raise TypeMismatch
    | Neq (e1, e2)    -> let (typ1, temp1, instrs1) = check_expr e1 vt
                        and (typ2, temp2, instrs2) = check_expr e2 vt
                        in
                            if typ1 = Matrix && typ2 = Matrix then
                                let temp = add_temp Boolean vt
                                in
                                    let instrs = instrs1 @ instrs2 @
                                                 [SMNE (temp, temp1, temp2)]
                                    in (Boolean, temp, instrs)
                            else if typ1 = String && typ2 = String then
                                let cmp_temp = add_temp Scalar vt
                                and temp = add_temp Boolean vt
                                in
                                    let instrs = instrs1 @ instrs2 @
                                                 [SCMP (cmp_temp, temp1, temp2);
                                                  SNE (temp, SVar cmp_temp, SLit 0.0)]
                                    in (Boolean, temp, instrs)
                            else if typ1 = typ2 then
                                let temp = add_temp Boolean vt
                                in
                                    let instrs = instrs1 @ instrs2 @
                                                 [SNE (temp, SVar temp1, SVar temp2)]
                                    in (Boolean, temp, instrs)
                            else
                                raise TypeMismatch
```

```
      | Lt (e1, e2)     -> let (typ1, temp1, instrs1) = check_expr e1 vt
                           and (typ2, temp2, instrs2) = check_expr e2 vt
                           in
                               if typ1 <> typ2 then
                                   raise TypeMismatch
                               else if typ1 = String then
                                   let cmp_temp = add_temp Scalar vt
                                   and temp = add_temp Boolean vt
                                   in
                                       let instrs = instrs1 @ instrs2 @
                                                    [SCMP (cmp_temp, temp1, temp2);
                                                     SLT (temp, SVar cmp_temp, SLit 0.0)]
                                       in (Boolean, temp, instrs)
                               else if typ2 = Scalar then
                                   let temp = add_temp Boolean vt
                                   in
                                       let instrs = instrs1 @ instrs2 @
                                                    [SLT (temp, SVar temp1, SVar temp2)]
                                       in (Boolean, temp, instrs)
                               else
                                   raise WrongDataType
      | Gt (e1, e2)     -> let (typ1, temp1, instrs1) = check_expr e1 vt
                           and (typ2, temp2, instrs2) = check_expr e2 vt
                           in
                               if typ1 <> typ2 then
                                   raise TypeMismatch
                               else if typ1 = String then
                                   let cmp_temp = add_temp Scalar vt
                                   and temp = add_temp Boolean vt
                                   in
                                       let instrs = instrs1 @ instrs2 @
                                                    [SCMP (cmp_temp, temp1, temp2);
                                                     SGT (temp, SVar cmp_temp, SLit 0.0)]
                                       in (Boolean, temp, instrs)
                               else if typ1 = Scalar then
                                   let temp = add_temp Boolean vt
                                   in
                                       let instrs = instrs1 @ instrs2 @
                                                    [SGT (temp, SVar temp1, SVar temp2)]
                                       in (Boolean, temp, instrs)
                               else
                                   raise WrongDataType
      | Le (e1, e2)     -> let (typ1, temp1, instrs1) = check_expr e1 vt
                           and (typ2, temp2, instrs2) = check_expr e2 vt
                           in
                               if typ1 <> typ2 then
                                   raise TypeMismatch
                               else if typ1 = String then
                                   let cmp_temp = add_temp Scalar vt
                                   and temp = add_temp Boolean vt
                                   in
                                       let instrs = instrs1 @ instrs2 @
                                                    [SCMP (cmp_temp, temp1, temp2);
                                                     SLE (temp, SVar cmp_temp, SLit 0.0)]
                                       in (Boolean, temp, instrs)
                               else if typ1 = Scalar then
                                   let temp = add_temp Boolean vt
                                   in
                                       let instrs = instrs1 @ instrs2 @
                                                    [SLE (temp, SVar temp1, SVar temp2)]
                                       in (Boolean, temp, instrs)
                               else
                                   raise WrongDataType
      | Ge (e1, e2)     -> let (typ1, temp1, instrs1) = check_expr e1 vt
                           and (typ2, temp2, instrs2) = check_expr e2 vt
                           in
```

```
                            if typ1 <> typ2 then
                                raise TypeMismatch
                            else if typ1 = String then
                                let cmp_temp = add_temp Scalar vt
                                and temp = add_temp Boolean vt
                                in
                                    let instrs = instrs1 @ instrs2 @
                                                 [SCMP (cmp_temp, temp1, temp2);
                                                  SGE (temp, SVar cmp_temp, SLit 0.0)]
                                    in (Boolean, temp, instrs)
                            else if typ1 = Scalar then
                                let temp = add_temp Boolean vt
                                in
                                    let instrs = instrs1 @ instrs2 @
                                                 [SGE (temp, SVar temp1, SVar temp2)]
                                    in (Boolean, temp, instrs)
                            else
                                raise WrongDataType
      | Not e           -> let (typ1, temp1, instrs1) = check_expr e vt
                           in
                            if typ1 <> Boolean then
                                raise WrongDataType
                            else
                                let temp = add_temp Boolean vt
                                in
                                    let instrs = instrs1 @ [NOT (temp, BVar temp1)]
                                    in (Boolean, temp, instrs)
      | And (e1, e2)    -> let (typ1, temp1, instrs1) = check_expr e1 vt
                           and (typ2, temp2, instrs2) = check_expr e2 vt
                           in
                            if typ1 <> Boolean || typ2 <> Boolean then
                                raise WrongDataType
                            else
                                let temp = add_temp Boolean vt
                                in
                                    let instrs = instrs1 @ instrs2 @
                                                 [AND (temp, BVar temp1, BVar temp2)]
                                    in (Boolean, temp, instrs)
      | Or (e1, e2)     -> let (typ1, temp1, instrs1) = check_expr e1 vt
                           and (typ2, temp2, instrs2) = check_expr e2 vt
                           in
                            if typ1 <> Boolean || typ2 <> Boolean then
                                raise WrongDataType
                            else
                                let temp = add_temp Boolean vt
                                in
                                    let instrs = instrs1 @ instrs2 @
                                                 [OR (temp, BVar temp1, BVar temp2)]
                                    in (Boolean, temp, instrs)
      | Trans e         -> let (typ1, temp1, instrs1) = check_expr e vt
                           in
                            if typ1 = Matrix then
                                let temp = add_temp Matrix vt
                                in
                                    let instrs = instrs1 @
                                                 [INIT temp;
                                                  TRAN (temp, temp1)]
                                    in (Matrix, temp, instrs)
                            else if typ1 = Scalar then
                                (Scalar, temp1, instrs1)
                            else
                                raise WrongDataType
      | SizeRows e      -> let (typ1, temp1, instrs1) = check_expr e vt
                           in
                            if typ1 <> Matrix then
                                raise WrongDataType
```

```
                                else
                                    let temp = add_temp Scalar vt
                                    in
                                        let instrs = instrs1 @
                                                     [ROWS (temp, temp1)]
                                        in (Scalar, temp, instrs)
      | SizeCols e      -> let (typ1, temp1, instrs1) = check_expr e vt
                           in
                                if typ1 <> Matrix then
                                    raise WrongDataType
                                else
                                    let temp = add_temp Scalar vt
                                    in
                                        let instrs = instrs1 @
                                                     [COLS (temp, temp1)]
                                        in (Scalar, temp, instrs)

and check_lvalue lv vt =
    match lv with
        Ident name             -> let n = find_variable name vt
                                  in
                                      let typ = get_variable_type n vt
                                      in (typ, n, [])
      | MAccess (name, e1, e2) -> let n = find_variable name vt
                                  in
                                      let typ0 = get_variable_type n vt
                                      and (typ1, temp1, instrs1) = check_expr e1 vt
                                      and (typ2, temp2, instrs2) = check_expr e2 vt
                                      in
                                          if typ0 <> Matrix || typ1 <> Scalar || typ2 <> Scalar then
                                              raise WrongDataType
                                          else
                                              let temp = add_temp Scalar vt
                                              in
                                                  let instrs = instrs1 @ instrs2 @
                                                               [GET (n, SVar temp1, SVar temp2, temp)]
                                                  in (Scalar, temp, instrs)
      | VAccess (name, e)      -> let n = find_variable name vt
                                  in
                                      let typ0 = get_variable_type n vt
                                      and (typ1, temp1, instrs1) = check_expr e vt
                                      in
                                          if typ0 <> Matrix || typ1 <> Scalar then
                                              raise WrongDataType
                                          else
                                              let temp = add_temp Scalar vt
                                              in
                                                  let instrs = instrs1 @
                                                               [GET (n, SVar temp1, SLit 0.0, temp)]
                                                  in (Scalar, temp, instrs)

let gen_init n typ =
    if typ = Matrix then
        [INIT n]
    else if typ = String then
        [SINI n]
    else
        []

let rec check_stmt s vt ls =
    match s with
        Assign (Ident name, e)             -> let (typr, tempr, instrsr) = check_expr e vt
                                              and n = find_variable name vt
                                              in
                                                  let typ = get_variable_type n vt
                                                  in
```

```
                                            if typ <> typr then
                                                raise TypeMismatch
                                            else if typ = Scalar then
                                                instrsr @ [SCAL (n, SVar tempr)]
                                            else if typ = Boolean then
                                                instrsr @ [BOOL (n, BVar tempr)]
                                            else if typ = String then
                                                instrsr @ [SCPY (n, tempr)]
                                            else if typ = Matrix then
                                                instrsr @ [SMUL (n, tempr, SLit 1.0)]
                                            else
                                                raise WrongDataType (* should never happen *)
     | Assign (MAccess (name, e1, e2), e) -> let (typr, tempr, instrsr) = check_expr e vt
                                             and n = find_variable name vt
                                             in
                                                 let typ0 = get_variable_type n vt
                                                 and (typ1, temp1, instrs1) = check_expr e1 vt
                                                 and (typ2, temp2, instrs2) = check_expr e2 vt
                                                 in
                                                     if typ0 <> Matrix || typ1 <> Scalar || typ2 <> Scalar ↙
    || typr <> Scalar then
                                                         raise WrongDataType
                                                     else
                                                         let instrs = instrs1 @ instrs2 @ instrsr @
[SET (n, SVar temp1, SVar temp2, SVar tempr)]
                                                         in instrs
     | Assign (VAccess (name, e1), e)     -> let (typr, tempr, instrsr) = check_expr e vt
                                             and n = find_variable name vt
                                             in
                                                 let typ0 = get_variable_type n vt
                                                 and (typ1, temp1, instrs1) = check_expr e1 vt
                                                 in
                                                     if typ0 <> Matrix || typ1 <> Scalar || typr <> Scalar ↙
    then
                                                         raise WrongDataType
                                                     else
                                                         let instrs = instrs1 @ instrsr @
[SET (n, SVar temp1, SLit 0.0, SVar tempr)]
                                                         in instrs
     | If (e, s1)                         -> let (typc, tempc, instrsc) = check_expr e vt
                                             in
                                                 if typc <> Boolean then
                                                     raise WrongDataType
                                                 else
                                                     let instrst = check_stmt s1 vt ls
                                                     and labend = add_label ls
                                                     in
                                                         instrsc @
                                                         [BRAF (BVar tempc, labend)] @
                                                         instrst @
                                                         [LABL labend]
     | IfElse (e, s1, s2)                 -> let (typc, tempc, instrsc) = check_expr e vt
                                             in
                                                 if typc <> Boolean then
                                                     raise WrongDataType
                                                 else
                                                     let instrst = check_stmt s1 vt ls
                                                     and instrse = check_stmt s2 vt ls
                                                     and labelse = add_label ls
                                                     and labend = add_label ls
                                                     in
                                                         instrsc @
                                                         [BRAF (BVar tempc, labelse)] @
                                                         instrst @
                                                         [JMP labend;
                                                          LABL labelse] @
```

```
                                                instrse @
                                                [LABL labend]
      | While (e, s1)         -> let (typc, tempc, instrsc) = check_expr e vt
                                 in
                                     if typc <> Boolean then
                                         raise WrongDataType
                                     else
                                         let instrsb = check_stmt s1 vt ls
                                         and labcheck = add_label ls
                                         and labend = add_label ls
                                         in
                                             [LABL labcheck] @
                                             instrsc @
                                             [BRAF (BVar tempc, labend)] @
                                             instrsb @
                                             [JMP labcheck;
                                              LABL labend]
      | Print e               -> let (typ0, temp0, instrs0) = check_expr e vt
                                 in
                                     if typ0 = String then
                                         instrs0 @ [PRINT temp0]
                                     else if typ0 = Scalar then
                                         let temp = add_temp String vt
                                         in
                                             instrs0 @
                                             [SINI temp;
                                              SFSC (temp, SVar temp0);
                                              PRINT temp]
                                     else if typ0 = Matrix then
                                         let temp = add_temp String vt
                                         in
                                             instrs0 @
                                             [SINI temp;
                                              SFMA (temp, temp0);
                                              PRINT temp]
                                     else
                                         raise WrongDataType
      | Dim (name, e1, e2)    -> let n = find_variable name vt
                                 in let typ0 = get_variable_type n vt
                                 and (typ1, temp1, instrs1) = check_expr e1 vt
                                 and (typ2, temp2, instrs2) = check_expr e2 vt
                                 in
                                     if typ0 <> Matrix || typ1 <> Scalar || typ2 <> Scalar then
                                         raise WrongDataType
                                     else
                                         instrs1 @ instrs2 @
                                         [RDIM (n, SVar temp1, SVar temp2)]
      | Decl (t, name)        -> let n = add_variable name t vt
                                 in
                                     gen_init n t
      | DeclInit (t, name, e) -> let n = add_variable name t vt
                                 and (typ1, temp1, instrs1) = check_expr e vt
                                 in
                                     if t <> typ1 then
                                         raise TypeMismatch
                                     else if t = String then
                                         instrs1 @ [SINI n; SCPY (n, temp1)]
                                     else if t = Matrix then
                                         instrs1 @ [INIT n; SMUL (n, temp1, SLit 1.0)]
                                     else if t = Scalar then
                                         instrs1 @ [SCAL (n, SVar temp1)]
                                     else if t = Boolean then
                                         instrs1 @ [BOOL (n, BVar temp1)]
                                     else
                                         raise WrongDataType (* should never happen *)
      | StmtList slist        -> check_prgm slist vt ls
```

```
and check_prgm slist vt ls =
    let helper s = check_stmt s vt ls
    and concat_lists l1 l2 = l1 @ l2
    in let list_of_lists = List.map helper slist
    in List.fold_left concat_lists [] list_of_lists
```

```
(* David Golub *)

open Ast
open Icode
open Labels
open Vars

exception InvalidMatrix
exception TypeMismatch
exception WrongDataType

val check_matrix : matrix -> vartable -> (int * iprog)
val check_expr : expr -> vartable -> (datatype * int * iprog)
val check_lvalue : lvalue -> vartable -> (datatype * int * iprog)
val check_stmt : stmt -> vartable -> labels -> iprog
val check_prgm : prgm -> vartable -> labels -> iprog
```

```
(* David Golub *)

open Ast

type variable = { name : string; typ : datatype }
type vartable = { mutable table : variable array; mutable next : int }

exception VariableNotFound of string
exception DuplicateDefinition of string

let new_vartable () = { table = [| |]; next = 0 }
let check_variable name vt =
    let rec helper n =
        if n < vt.next then
            if vt.table.(n).name = name then
                true
            else
                helper (n + 1)
        else
            false
     in helper 0
let add_variable name typ vt =
    if check_variable name vt then
        raise (DuplicateDefinition name)
    else
        let varno = vt.next
         in vt.table <- Array.append vt.table [| { name = name; typ = typ } |];
            vt.next <- varno + 1;
            varno
let add_temp typ vt =
    let name = "_temp" ^ string_of_int vt.next
    in add_variable name typ vt
let find_variable name vt =
    let rec helper n =
        if n < vt.next then
            if vt.table.(n).name = name then
                n
            else
                helper (n + 1)
        else
            raise (VariableNotFound name)
     in helper 0
let get_variable_name n vt =
    vt.table.(n).name
let get_variable_type n vt =
    vt.table.(n).typ
let gencpp_datatype typ =
    match typ with
        Scalar  -> "Scalar"
      | Matrix  -> "Matrix"
      | String  -> "String"
      | Boolean -> "Boolean"
let gencpp_vartable vt =
    let rec helper n =
        if n < 0 then
            ""
        else
            "\t" ^ (gencpp_datatype vt.table.(n).typ) ^ " " ^ vt.table.(n).name ^ ";\n" ^ (helper (n - 1))
    in
        helper (vt.next - 1)
```

```
(* David Golub *)

open Ast

type vartable

exception VariableNotFound of string
exception DuplicateDefinition of string

val new_vartable : unit -> vartable
val check_variable : string -> vartable -> bool
val add_variable : string -> datatype -> vartable -> int
val add_temp : datatype -> vartable -> int
val find_variable : string -> vartable -> int
val get_variable_name : int -> vartable -> string
val get_variable_type : int -> vartable -> datatype
val gencpp_vartable : vartable -> string
```

```
matrix A = { 1, 0; 0, 1 };
matrix B = { 0, 1; 1, 0 };
print A + B;
```

```
matrix A = { 1, 2, 3; 4, 5, 6 };
print A ^ 2;
```

```
matrix A = { 0, 0; 0, 0 };
print A[0, 2];
```

```
matrix A = { 1, 2; 3, 4 };
print A[0, 1];
print "\n";
A[1, 1] = 5;
print A;
```

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

```
scalar x = 1;
if(x < 5) {
    print "x < 5\n";
} else {
    print "Something else\n";
}
if(x >= 5) {
    print "Something else\n";
} else {
    print "x < 5\n";
}
```

```
scalar x = 1;
if(x < 5) {
    print "x < 5";
}
```

```
DEPTH = ..

TESTS = \
    hello \
    addmat \
    submat \
    scalmul \
    mulmat \
    elemacc \
    vectacc \
    iftest \
    ifelse \
    matsize \
    matpow \
    badpow \
    rowbound \
    colbound \
    redim \
    trans \
    tutorial \
    vel

all : $(TESTS)

$(TESTS) :
    $(DEPTH)\lame < $@.lam > $@.cpp
    cl -nologo -EHsc -I$(DEPTH) $@.cpp
    $@ > output\$@.out
```

```
DEPTH = ..

TESTS = \
    hello \
    addmat \
    submat \
    scalmul \
    mulmat \
    elemacc \
    vectacc \
    iftest \
    ifelse \
    matsize \
    matpow \
    badpow \
    rowbound \
    colbound \
    redim \
    trans \
    tutorial

all : $(TESTS)

$(TESTS) :
    $(DEPTH)\lame < $@.lam > $@.cpp
    cl -nologo -EHsc -I$(DEPTH) $@.cpp
    $@ > output\$@.out
```

```
matrix A = { 1, 2; 3, 4 };
print A ^ 2;
```

```
matrix A = { 1, 2, 3; 4, 5, 6 };
print size_rows A + "\n";
print size_cols A + "\n";
```

```
matrix A = { 0, 1; 1, 0 };
matrix B = { 1, 1; 1, 1 };
print A * B;
```

```
matrix A = { 1, 0; 0, 1 };
print A;
```

```
matrix A = {
    1, 2, 3;
    4, 5, 6;
    7, 8, 9
};
dim A[2, 2];
print A;
dim A[3, 3];
print A;
```

```
matrix A = { 0, 0; 0, 0 };
print A[2, 0];
```

```
matrix A = { 1, 0; 0, 1 };
print A * 2;
```

```
matrix A = { 1, 2; 3, 4 };
print A';
```

```
matrix A = { 3, 1; 9, 4 };
matrix B = { 3; 6 };
matrix X;

print "\nSolving system of simultaneous linear equations:\n\n";
print A[0,0] + " x1 + " + A[0,1] + " x2 = " + B[0] + "\n";
print A[1,0] + " x1 + " + A[1,1] + " x2 = " + B[1] + "\n";

print "\nA = \n" + A + "\n";
print "\nB = \n" + B + "\n";

scalar det_of_A = A[0,0]*A[1,1] - A[0,1]*A[1,0];

print "\nDeterminant(A) = " + det_of_A + "\n";

if(det_of_A != 0) {

    matrix inv_of_A;

    inv_of_A [0,0] = A[1,1];
    inv_of_A [0,1] = -1*A[0,1];
    inv_of_A [1,0] = -1*A[1,0];
    inv_of_A [1,1] = A[0,0];
    inv_of_A = inv_of_A / det_of_A;
    X = inv_of_A * B;

    print "\nInverse(A) = \n" + inv_of_A + "\n";
    print "X = Inverse(A) * B = \n" + X + "\n";
    print "Solution:\n";
    print "x1 = " + X[0] + "\n";
    print "x2 = " + X[1] + "\n";

} else {

    print "A is a singular matrix and its inverse does not exist. So, solution not found.\n";

}
```

```
matrix A = { 1; 2; 3; 4; 5; 6; 7; 8; 9 };
scalar i = 0;
while(i < 6) {
    print A[i];
    print "\n";
    i = i + 1;
}
```

```
matrix vt = { 50, 0; 0, 50 };
matrix a = { 0, 0; 0, 9.8 };

matrix vf = {0, 0; 0, 0};

scalar t = 0;

while(t < 100)
{

    vf = vt + (1/2) * -1 * a * t * t;

    t = t + 1;

    print t + ": " + vf[1,1] + "\n";

}
```