JME:
A Lightweight Statistical Language

Daniel Gordon

August 12, 2014

Contents

Chapter 1

Introduction

JME (pronounced jay+me) is a lightweight language that allows programmers to easily perform statistic computations on tabular data as part of data analysis. The language is designed to perform calculations and operations on primitive data structures like a vector or matrix, as well as a higher-level structure like a map. JME is designed as a scripting tool for single time analysis of a data set. JME has several commonly used statistical measures built in but will allow the user to express any number of statistical analysis algorithms through the creation of functions.

1.1 Goals of JME

The high level goal of JME is a language that is powerful yet easy to learn.

1.1.1 Simple yet Powerful

The syntax of JME is designed to be relatively simple. Declaring variables, creating functions and initializing data structures require a minimum amount of syntax allowing developers to do more while writing less. The goal is to minimize the barrier to entry for new users as well as developing programs faster.

1.1.2 Dynamically and Weakly Typed

Part of making the language simple is the decision to make JME a dynamically typed language similar to other dynamically typed languages of Ruby and Javascript. Variables do not need to be initialized with a type, simply assigning a variable to a value will let the compiler know what type that variable is. Variables can then be re-assigned to different types in the lifetime of the program without throwing an exception. Like Javascript, JME is weakly typed, meaning that once a variable is assigned to a type it can potentially be mixed with other types in the same expression. This also means that some operators and functions are overloaded to handle different data types appropriately. While this has the advantage of minimizing the syntax and knowledge required to begin programming in JME, it has the disadvantage of producing run-time errors. JME has a built in function for each datatype that accepts an expression and returns a Boolean value if the given expression is of that datatype.

Chapter 2

Tutorial

2.1 Getting Started

A JME program is written in a single file with a **.jme** extension. All JME programs start with a **main()** function. Within the **main()** function programming statements are written to perform computations. In addition outside of the **main()** function any number of other functions can be declared and used.

2.2 Compiling and Running

To compile the JME executable, from the command line cd into the jme directory and run the command *make*.

```
cd <pathto>/jme
make
```

JME programs can be written in any text editor. Once written, JME programs are compiled and executed from the command line using the **jme** executable. A JME program is piped into the **jme** executable as standard input and the jme executable will compile and execute the code. For example if we wanted to run the JME program called "helloworld.jme" the command line would look something like this:

```
./jme < helloworld.jme
Hello World
```

Note the above example assumes that the "helloworld.jme" file is located in the same directory as the **jme** executable.

2.2.1 Command Line Arguments

Several additional command line arguments can be applied when executing a JME program. These arguments can be useful for debugging a program or deciphering how the compiler is behaving:

-a — Outputs the abstract syntax tree of the program generated by the JME parser

-b — Outputs the bytecode representation of the program that is generated by the JME compiler

-c — Compiles the JME program and executes it. This is the default behavior if no arguments are added.

-wo — A secondary argument that can be combined with any of the above arguments. This will execute the JME program without the standard library functions (defined in full later in the document). Turning off the standard library can be very useful when debugging a program.

2.3 Hello World Example

As a first example of JME programming we will use the standard Hello World and show how to print the message "Hello World" to the screen:

```

main() {
  msg = "Hello World";
  print(msg);
}

/*prints Hello World */
```

In the above example the String literal "Hello World" is assigned to the identifier *msg*. The standard library **print()** function is called to print *msg*.

2.4 Finding the Average of a Set

A slightly more complex example is finding the average or mean of a given set of numbers. To do this we will use a Vector data structure to hold a set of numbers. We will also define a function avg to compute the average of the set. By creating a function we can then re-use the same code to compute the average for additional data sets:

```
avg(vect) {
 total = 0;
 for (i = 0 ; i < width(vect) ; i = i + 1) {
     total = total + vect[i];
   }
   return total / width(vect);
}

main()
{
a = [0, 1, 2,3, 4, 5,6, 7,8, 9, 10];
b = [9, 12, 5, 4, 12, 7.1, 8, 11, 20.4, 3, 7, 4, 1.2, 5, 4, 10,
    9, 6, 9, 4];

print(avg(a));
/* prints 5 */
print(avg(b));
/* prints 7.535 */
}
```

The **avg** function takes in a Vector type as a single parameter identified as *vect*. A **for** loop is then used to sum the numbers within the passed in Vector. The built in function **width()** is used to determine the total number of elements within the given vector. Finally the sum total is divided by the number of elements in the vector and the result is returned. In the **main** function two vectors are initialized and assigned to identifiers a and b respectively. The **avg** function is called for both a and b vectors and the results are printed.

2.5 More Examples

2.5.1 Matrix Example

JME also supports the creation of Matrices. The syntax for declaring a matrix is very similar to that of declaring a vector. The only difference is that the rows are separated by a semicolon:

```
main() {
    /* initialize 2x4 matrix */
    a = [1,2; 3,4; 5,6; 7,8];

    print(width(a));
```

```
    /* prints 2 */

    print(height(a));
    /* prints 4 */

    print(a[1][1]);
    /* prints value of 2nd row 2nd column: 4 */

    a[0][0] = 99;
    /* update first element of matrix a to 99 */

    print(a);

    /* prints
        [99, 2;
         3, 4;
         5,6;
         7,8]
     */
}
```

2.5.2 Map Example

JME has a built in map data structure. The map data structure stores a series of key-value pairs, where the key is a string and the value can be any data type. Accessing individual values of the map can be done with the key. A map is initialized with vertical bars and the key is pointed to the value using the equals sign and the greater than operator.

```
main()
{
 foo = "foo";
 map = new | "a" => 5, "b" =>3, foo => "barr" , "zz" => "fist"
    |;
 map["c"] = map["a"] + map["b"];
 map["a"] = 32;

 print(map);

 /* prints |zz=>fist , foo=>barr , c=>8, b=>3, a=>32 | */
}
```

Chapter 3

Language Reference Manual

3.1 High Level Program Structure

At a high level all JME programs have a main body of the program which includes a list of statements, which comprise of expressions and various control structures. In addition to the main body of the program, developers are free to define their own functions which can then be called and re-used from the main body of the program. There is also a variety of built in functions that can be leveraged. Formal definition of control structures and functions are laid out below.

3.2 Lexical Conventions

The grammar of the language will be defined in this section. Symbols in italics are to be treated as non-terminals, and symbols in bold are terminals. Standard regular expressions are used notate the lexical pattern expected by the compiler.

The JME language contains a set of tokens that includes identifiers, keywords, constants, expression operators and separators. The compiler ignores White space, such as spaces, newlines, and tabs, along with comments.

There are 5 types of tokens defined: identifiers, keywords, constants, separators, and expression operators. Tokens are separated by whitespace. Token generation is greedy meaning that the next token is defined as the longest string of characters that could possibly constitute a token.

3.2.1 White space

Whitespace is defined as spaces, newlines, horizontal tabs, and line terminators. Whitespace is required to separate adjacent tokens. Whitespace is ignored by the compiler.

3.2.2 Comments

The JME language supports C-style like comments. A comment begins with the characters `/*` and ends with the characters `*/`. Inside the comment block all characters are accepted, with the exception of `*/` which would terminate the comment block. Comment blocks can span multiple lines.

3.2.3 Identifiers

Identifiers are defined as a sequence of letters or numbers. Identifiers must start with a letter. There is no limit to the length of an identifier. Identifiers are case sensitive meaning that two identifiers with the same alphanumeric character string but with differing cases will be seen as two distinct identifiers.

$$\textit{Identifier} \rightarrow \textit{letter}\ (\ \textit{letter} \mid \textit{digit}\,)*$$

3.2.4 Keywords

The following are reserved for use as keywords, and may not be used otherwise:

else	**if**	**true**
false	**new**	**var**
for	**return**	**while**

3.2.5 Constants

There are four primitive data types that can be expressed as literals in JME:

Integers

An integer literal is a sequence of digits 0-9 respectively.

$$\textit{Integer} \rightarrow [\,\text{‘}\mathbf{0}\text{‘} - \text{‘}\mathbf{9}\text{‘}\,]+$$

Floating Point Numbers

Floating constants consist of three parts, an integer part, a decimal point and a fraction part. The fraction part and the integer part consist of a sequence of digits. The fraction part is required however the integer part is optional. Floating point numbers can optionally be signed with a minus sign to indicate the number is negative. Every floating constant is taken to be double precision.

$$\mathit{Float} \rightarrow [\text{`}\mathbf{-}\text{`}]?\,[\text{`}\mathbf{0}\text{`} - \text{`}\mathbf{9}\text{`}]*\ [\text{`}\mathbf{.}\text{`}]\,[\text{`}\mathbf{0}\text{`} - \text{`}\mathbf{9}\text{`}]+$$

Strings

A String is a sequence of characters enclosed in double quotes. Strings can contain any combination of characters except for newline, break line, and double quote terminals. The empty string, containing no characters, can be defined as two double quotes with no space: **“”**.

$$\mathit{String} \rightarrow \text{`}\mathbf{``}\text{`}\ [\hat{}\ \backslash\mathbf{r}\,\backslash\mathbf{n}\ \mathbf{``}]*\ \text{`}\mathbf{''}\text{`}$$

Boolean

A Boolean constant is a logical data type and can have two possible values: true or false. Boolean constants are defined using the reserved keywords **true** or **false**.

$$\mathit{Boolean} \rightarrow \text{``}\mathbf{true}\text{``}\,|\,\text{``}\mathbf{false}\text{``}$$

Null

Null represents the empty type. For variables or data structures with uninitialized elements the Null type is used.

null

3.2.6 Separators

The following characters are used as separators and serve to define structure within the language:

{ } ( ) [ ] ; . ,

3.2.7 Operators

The following operators are used when defining mathematical or logical expressions:

+ - * /	Plus, Minus, Multiply, Divide
=	Assignment to an Identifier
==	Logical equals
!=	Logical not equals
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
=>	Assignment within a Map
ˆ	Exponent
%	Remainder operator

3.3 Expressions and Operators

Expressions can use operators to define a mathematical or logical computation. Parentheses can be used to explicitly define an order of operation. Otherwise the order of evaluation of an expression is undefined. Multiple expressions can be separated using the semicolon. Expressions can be defined using constants or identifiers or a mix of both.

3.3.1 Primary Expressions

An expression does not necessarily have to include an operator. It can be defined as a single constant or a single identifier. This can be a numeric constant (float or integer), a Boolean constant (true or false), or a string constant.

$$\mathit{Expression} \rightarrow (\mathit{constant}|\mathit{identifier});$$

3.3.2 Binary Arithmetic Expressions

A binary arithmetic expression must include at least two constants or identifiers separated by a mathematical operator. The expressions can be a mix of integer or floating point numbers, however the integer will always be converted to a floating point number when performing the calculation.

expr ˆ expr

Returns the result of the left expression raised to the power of the right expression. The right expression must evaluate to an integer. The exponent operator takes precedence over other mathematical operators. The result will always be a floating point number.

expr * expr

Returns the product of two expressions. Multiplication takes precedence over subtraction and addition. Multiplication and division operators are evaluated from left to right.

expr / expr

Returns the quotient of the left expression divided by the right expression. If two integers are divided and the right expression does not divide evenly into the left expression then the result is round down to the nearest integer. Division takes precedence over subtraction and addition. Multiplication and division operators are evaluated from left to right.

expr % expr

Returns the remainder of the left expression divided by the right expression. Remainder operation takes precedence over addition and subtraction. Remainder operation is evaluated at the same level as multiplication and division.

expr - expr

Returns the result of the right expression subtracted from the left expression.

expr + expr

Returns the result of the left expression added to the right expression.

3.3.3 Relational Expressions

A relational expression evaluates two expressions based on the relational operator and returns a Boolean value (true or false). String, Boolean, integer and floating types can be compared. Same types must be compared, including comparsions of integer and floating point numbers.

expr == expr

This operator evaluates the equality of the two expressions.

expr != expr

This operator evaluates if the two expressions are unequal.

expr > expr

This operator evaluates to true if the left expression is greater than the right expression.

expr >= expr

This operator evaluates to true if the left expression is greater than or equal to the right expression.

expr < expr

This operator evaluates to true if the left expression is less than the right expression.

expr <= expr

This operator evaluates to true if the left expression is less than or equal to the right expression.

3.4 Variables and Scope

3.4.1 Assignment Expression

$$Identifier = expr$$

The equals operator is right associative and used to assign an Identifier to a literal or a data structure. The expression is evaluated and the resulting type is assigned to the Identifier. The Identifier can then be used as a variable in subsequent lines of code. The same identifier can be re-assigned to a different value or even a different type if another assignment statement takes place. JME is a dynamically typed language so when initializing a variable there is no need to explicitly define the type.

3.4.2 Local Variables

Variables when first declared inside of a function have a lifespan only for that function and cannot be read outside of the function. This applies to the **main()** function as well as any user defined functions. Parameters for functions are considered local variables whose lifespan is limited to the function itself.

3.4.3 Global Variables

JME also supports the use of global variables. These variables must be identified outside of the main function and any user defined functions. Global variables are defined by using the keyword **var** and are initialized as null. Global variables cannot be initialized with a starting value and can only be assigned a type from within a function. Once defined a global variable can be used from within any function. The only caveat with global variables is that once defined, a local variable with the same identifier can never be explicitly defined.

$$globalVar \rightarrow \textbf{var} identifier\textbf{;}$$

3.5 Datatypes

Along with the primitive types of Integer, Float, Boolean and String there are three datatypes that can be defined in JME: Vectors, Matrices, and Maps.

3.6 Vectors

Vectors are the simplest data structure in the JME language and are defined as a one-dimensional list with a fixed size. Vectors can include either float or integers.

3.6.1 Initialization of Vectors

Vectors can be initialized using brackets and a comma-separated list of expressions. The expressions used to initialize a vector can be any valid expression including literal values, identifiers, binary arithmetic, or function calls. Vectors can be initialized with any number of values, however once initialized the vector size is fixed.

$$vector \rightarrow \Big[expr_1\textbf{,}expr_2\textbf{,}...\textbf{,}expr_n\Big]$$

An empty vector of a specified size can be initialized by using the **new** keyword. When creating an empty vector all of the elements are initialized to **null**. An integer literal can be used to specify the starting size of the vector. An expression can also be used to specify the size, provided that the expression evaluates to an integer.

$$emptyVector \rightarrow \textbf{new}\Big[expr^1\Big]$$

3.6.2 Elements of a Vector

Elements within the vector are numbered from 0 to n-1 where n is the length of the vector. Individual elements of a vector assigned to an identifier can be accessed by using the name of the identifier followed by bracket index notation with an integer expression that represents the desired element. Attempting to access an element outside the bounds of the vector will throw an exception.

$$vElement \rightarrow Identifier\Big[expr^2\Big]$$

Vector elements are mutable and individual elements can be updated. Updating individual elements of a vector can be accomplished using the same notation as accessing an element and the assignment operator.

$$Identifier\Big[expr\Big] = expr_{new}$$

3.6.3 Vectors and Scalars

A scalar can be applied to a vector using mulitplication and division operators

3.6.4 Vector width

The built in function **width()** will return an integer value representing the number of elements within a given vector. The built in function **height()** can also be applied to a vector but will always return an integer value of 1.

$$numElements = \textbf{width}(vector)$$

[1] Expression used to initialize empty vector must evaluate to an integer literal
[2] Expression used to access a vector element must evaluate to an integer literal

3.6.5 Vector Example

```
a = [1,2,3];
print(width(a));
// prints 3

print(height(a));
//prints 1
```

3.7 Matrices

Matrices are another data structure within JME, defined as a two-dimensional list with a fixed size of columns and rows. Like Vectors, Matrices elements can be either float or integers. Matrices can have any number of columns and rows; the only requirement is that the number of columns in each row is consistent throughout the matrix.

3.7.1 Matrix Initialization

Matrices are initialized with a similar syntax as Vectors using brackets and a comma-separated list of expressions where each expression represents a column. The semi colon **;** character is added to separate rows.

$$matrix \rightarrow \Big[expr_{1_1} \textbf{,} expr_{1_2} \textbf{,} ... \textbf{,} expr_{1_n} \textbf{;}\ expr_{n_1} \textbf{,} expr_{n_2} \textbf{,} ... \textbf{,} expr_{n_n}\Big]$$

An empty Matrix can be initialized with a given set of dimensions with two integer expressions enclosed in brackets and the **new** keyword. When creating an empty matrix all elements are initialized to **null**. The first integer expression defines the number of rows within the matrix. The second integer expression defines the number of columns within each row of the matrix. The two expressions can be integer literals or other type of expression but must evaluate to an integer.

$$emptyMatrix \rightarrow \textbf{new}\Big[expr_{row}\Big]\Big[expr_{col}\Big]$$

3.7.2 Elements of a Matrix

Each row of a matrix is indexed 0 to n -1 where n is the total number of rows. Each column within a row is indexed 0 to n -1 where n is the total number of columns within a row. Individual elements of a matrix can be accessed using bracket notation and two integer expressions where the first expression indicates the row index and the second expression indicates the column index.

$$matrixElement \rightarrow Identifier\big[expr_{row}\big]\big[expr_{col}\big]$$

Matrix elements are mutable and individual elements can be updated. Updating individual elements of a matrix can be accomplished using the same notation as accessing a matrix element and the assignment operator.

$$Identifier\big[expr_{row}\big]\big[expr_{col}\big] = expr_{new}$$

3.7.3 Matrices Height and Width

Matrix dimensions can be determined using the built in function **height()** and **width()**. The **height()** returns an integer value of the number rows within the given matrix. The **width()** function returns an integer value of the number of columns within a given row for the given matrix.

3.7.4 Matrices and Scalars

All elements of a matrix can be updated by a numeric scalar by using standard binary operators.

3.8 Maps

A map is a higher-level data structure consisting of multiple key-value pairs. The key must always evaluate to a string but the value can be any of the primitive data types such as an integer, float, Boolean or String. Values can also be Vectors or Matrices.

3.8.1 Initialization of a Map

A map is initialized with a comma-separated list of key-value pairs surrounded by vertical bars. The keyword **new** is required before the first vertical bar. The key can be a string literal or an expression that evaluates to a string. The value can be any expression that resolves to a type. The assignment of the key to a value is done using an equals operator and a greater than operator.

$$map \rightarrow \textbf{new}|expr_{key_1} => expr_{val_1}, ...expr_{key_n} => expr_{val_n}|$$

3.8.2 Accessing Values of a Map

The keys of a map are used to access individual values within the map itself. The identifier followed by bracket notation with the key as a string literal or string expression will return the value associated with that key. If the key does not exist in the current map null is returned.

$$mapValue \rightarrow identifier[expr_{key}]$$

Updating a Map

Once a map has been initialized additional key-value pairs can be added using the assignment operator and the same notation for accessing a value. The identifier assigned to the map followed by a string expression representing the key surrounded by brackets is written on the left side of the equals operator. On the right side the value the key should point to is written. The same notation can be used to update existing keys within the current map.

$$identifier[expr_{key}] = expr_{val}$$

Determine Existence of a Key in a Map

To determine if a key exists in a map the built in function **mhas()** can be used. The **mhas()** function accepts two arguments, the first being the identifier of the map and the second being an expression of the key in question. The **mhas()** returns a Boolean **true** if the value exists, otherwise **false**.

$$Boolean = \mathbf{mhas}(identifier, expr_{key})$$

3.9 Control Structures

3.9.1 Statements

Statements are defined as any expression or combination of expressions terminated with a semicolon. Unless specified statements are executed sequentially.

$$Statement \rightarrow expression;$$

3.9.2 Conditional Statement

Conditional statements use a boolean expression to determine the execution of a given block of code. Conditional statements use the keyword **if** followed by a Boolean expression surrounded by parenthesis and a series of statements surrounded by curly braces. If the Boolean expression evaluates to **true** the statements are executed, otherwise the statements are skipped. An optional set of statements prefixed with the keyword **else** and surrounded by curly braces can be added to the condition. The statements below **else** are executed if the initial Boolean expression evaluates to **false**.

$$\textbf{if(}expr_{bool}\textbf{)}\ \pmb{\{}\ statements\ \pmb{\}}\ (\textbf{else}\ \pmb{\{}\ statements\ \pmb{\}}\ )?$$

3.9.3 While Statement

The **while** statement is a looping structure uses the keyword **while** immediately followed by a Boolean expression surrounded by parenthesis and a block of statements surrounded by a curly brace. The block of statements are continuously executed until the initial Boolean expression evaluates to **false**.

$$\textbf{while}\ (expr_{bool})\pmb{\{}statements\pmb{\}}$$

3.9.4 For Statement

The **for** statement is another looping structure that uses the keyword **for** followed by three statements surrounded by parenthesis and a block of statements surrounded by curly braces. The first statement sets an identifier to an initial value, the second statement is a Boolean expression involving the identifier, and the third statement modifies the value of the identifier. The block of statements are executed in a loop as long as the second statement evaluates to true. For each iteration of the loop the third statement is executed immediately following the block of statements.

$$\textbf{for(}\ expr_1; expr_2; expr_3; \textbf{)}\pmb{\{}statementblock\pmb{\}}$$

3.9.5 Functions

Functions are comprised of a series of statements, can accept a number of arguments, and can return a value. Functions, once defined, can then be called as part of other statements.

Function Definition

Functions are defined with an identifier followed by a block of statements surrounded by curly braces. The identifier represents the name of the function and must be unique throughout the program. One or more identifiers can be added as parameters separated by a comma surrounded by parentheses. The set of identifiers or parameters represent values that can passed into the function. The curly braces mark the beginning and end of the function. Functions can contain any number of statements. Only parameters and identifiers local to the function can be used within the function itself. JME does not support nested functions.

Optionally the last statement of the function can contain the keyword **return** followed by an expression. The value of the expression is then returned to the original function call.

$$identifier\textbf{(}(identifer_1\textbf{,}identifier_2\textbf{,}...identifier_n)*\textbf{)}\ \textbf{\{}$$

$$statements$$

$$(\textbf{return}\ expr\textbf{;})?\textbf{\}}$$

Function Call

Functions, once defined can be called from the main function or from other functions as part of an expression or statement. Functions are called by using the identifier that defined the function followed by parentheses and any arguments that need to be passed in as parameters. JME supports recursion, so a function can call itself from within the statement block.

$$functionCall \rightarrow identifier\textbf{(}(parameter_1\textbf{,}parameter_2\textbf{,}...parameter_n)*\textbf{)}$$

3.9.6 Main()

The **main()** function is the entry point for all JME programs and is the first to be called. The **main()** function does not take any arguments and is executed just like any other function. From the **main()** function other functions can be called, but the **main()** function cannot be called by other functions. All JME programs must contain a **main()** function or the program will not be executed.

3.10 Built In Functions and Standard Library

JME has several functions built into the compiler. These functions such as **print()** are an inherit part of the JME compiler and are defined in the source code for JME. The Built in Functions are listed in full in Appendix A of this document. The Standard Library is a handful of functions for commonly used statistical measures. These functions are defined outside of the JME executable, are written in the JME language and can be updated by the developer if he/she chooses. The Standard Library functions are listed in full in Appendix B.

Chapter 4

Project Plan

The JME language will use an iterative approach to development, following a loosely based agile methodology. Language features will be broken down into small pieces and developed from end to end.

4.1 Team Responsibilities

The team is made up of a single developer, Daniel Gordon. Daniel is responsible for all aspects of the project including architecture, parser, grammar, and compiler. Daniel also is responsible for quality assurance and testing, as well as writing all documentation and reports associated with the project.

4.2 Project Timeline

The following deadlines were set as key milestones in the development cycle.

06-11-14	Language proposal
07-02-14	Language Reference Manual creation
07-07-14	Initial code commit and base unit tests
07-14-14	Primitive datatype: Integers, Floats, Boolean, and Strings
07-21-14	Vector datatype support
07-28-14	Matrix and Map datatype support
08-03-14	Standard Library functions
08-10-14	Code freeze and regression testing
08-18-14	Project report creation
08-22-14	Project submission

4.3 Software Development Environment

The project will be developed on Mac os (Linux) using OCaml, version 4.01.0. The text editor used to edit source files is Brackets: http://brackets.io/. Github.com will be used for Source Control Management: https://github.com/dgordon86/jme.

A Makefile will be created to compile all of the Ocaml source files and build the jme executable. A shell script will be used to run all unit tests.

4.4 Project Log

07-11-14	microc starter code (initial commit)
07-12-14	modified to use jme
07-12-14	deleted microc.ml
07-12-14	float data type, refactored code to store datatypes as own Type
07-12-14	modified test cases
07-12-14	boolean data type
07-13-14	updated binary operations
07-15-14	removed failed test
07-15-14	boolean operations now using true false instead of 0 or 1
07-20-14	added string support
07-20-14	first rendition of vector support
07-20-14	accessing individual elements of a vector
07-20-14	started code to modify individual elements of vector
07-21-14	added exponent operator
07-21-14	better test case for order of operations
07-21-14	added sqrt functionality
07-23-14	removed stdlib, switched backend datatype to array
07-24-14	vector assignment
07-25-14	testing vectors and standard deviation
07-27-14	modified floats to use negatives
07-27-14	no use var for local variables, added as they are seen
07-28-14	code to initialize empty vector structure
07-31-14	added matrix support and capabilities
08-11-14	added initial map functionality
08-11-14	moved stdlib to util

Chapter 5

Architectural Design

5.1 High Level Design

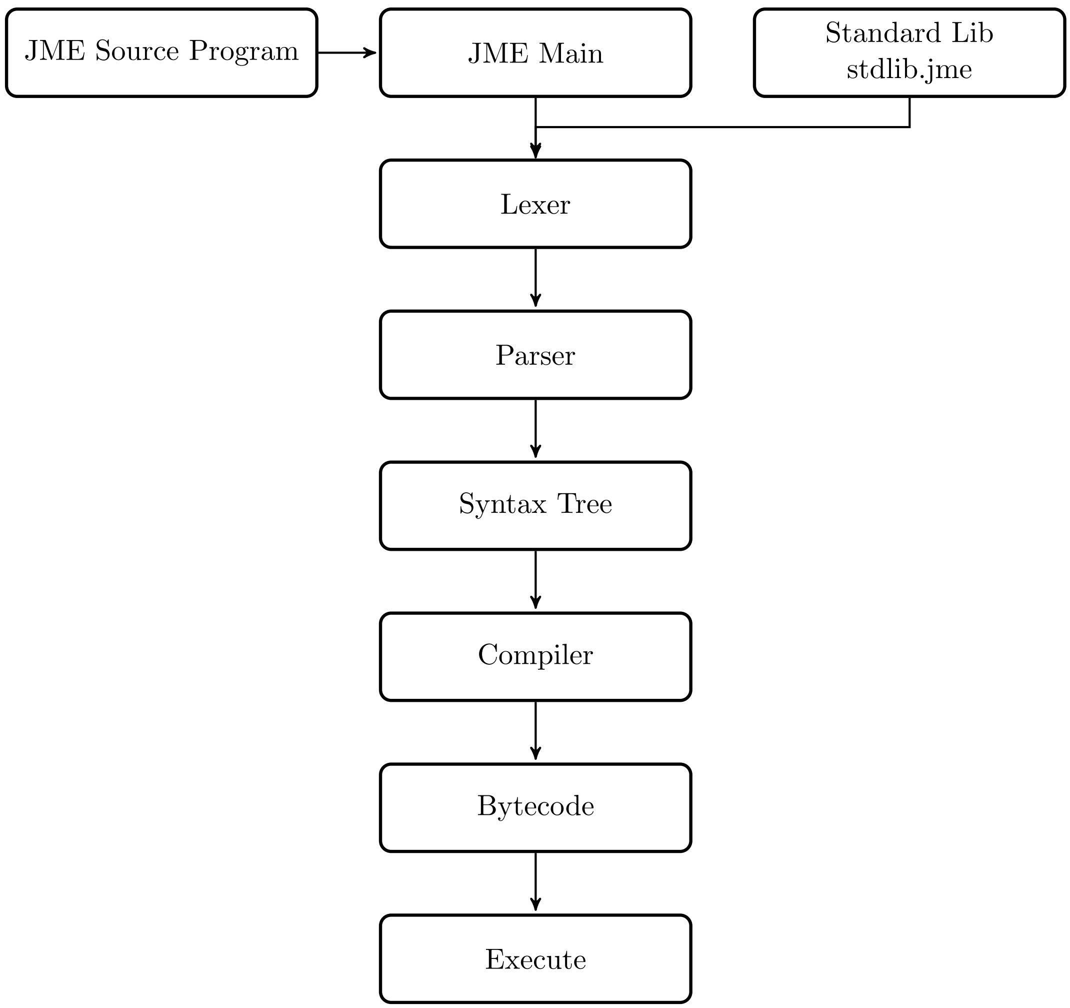

Figure 5.1 a high level architecture of the JME compiler.

5.2 Detailed Design

The JME compiler is made up of several components commonly found in compiler designs: lexer, parser, syntax tree, compiler, intermediary representation, and executable. The relationship between these pieces are illustrated in figure 5.1.

5.2.1 JME Source Program

The input to the JME compiler is the jme source program with a suffix of **.jme**.

5.2.2 JME Main

The JME Main component is the first entry point into the compiler. The source file for the main component is **jme.ml**. jme.ml evaluates the command line arguments and grabs the source program from standard input. The jme.ml then loads the standard library and calls the lexer, parser, abstract syntax tree, compiler and finally executes the code. Various command line arguments, defined in section 2.2.1, can be added to indicate to the compiler to run only a portion of the compiler, such as only generating the abstract syntax tree.

5.2.3 Standard Library

The Standard Library is defined in the source file **stdlib.jme**. The Standard Library consists of several common statistical functions. The Standard Library is meant to minimize the amount of code developers need to write in order to perform statistical analysis. Note the Standard Library differs from built in functions. The rationale for having the standard library in an external file was to make maintenance easier and allow developers to add their own functions to be re-used across JME programs. Since the Standard Library is in it's own file the JME compiler does not need to be re-built in order to use a new Standard Library function. The process of code linking is fairly basic and simply involves appending the JME source file with the Standard Library file and passing them both to the Lexer. This part of the architecture should be improved in future iterations.

5.2.4 Lexer and Parser

Ocamllex is used as a lexical analyzer for the source program. The JME source file for the lexical analyzer is **lexer.mll** which contains a series or regular expressions to tokenize the source file. Ocamlyacc is used to produce a parser for the JME grammar based on the lexical analyzer produced by Ocamllex. The input file that is used by Ocamlyacc is **parser.mly**

5.2.5 Abstract Syntax Tree

The Parser component parses the source input file and produces an Abstract Syntax Tree. The Abstract Syntax Tree is defined in **ast.ml**.

5.2.6 Compiler

The compiler component of the JME architecture is defined in the source file **compile.ml**. The compiler takes an Abstract Syntax Tree as input and works to produce an intermediary representation of the source input that can be quickly executed. The compiler traverses the Abstract Syntax Tree, defining declarations of variables, use of data types, and declarations of functions to transform the source file into JME bytecode representation. The bytecode used by JME is defined in the file **bytecode.ml**.

5.2.7 Execute

The execute component of the JME architecture executes the bytecode generated by the compiler. The execute component is defined in the source file **execute.ml**. The execute component uses a stack based implementation to execute the bytecode.

5.2.8 Other Files

Several other files are found in the JME source and are defined below:

datatypes.ml — the data types used in the JME language are defined along with overloaded operators for handling mixed datatypes

util.ml — a utility class for large blocks of code. Part of this class is a function that transforms an external file into a String. This is used to combine the source file and the standard library. This code is credited to open source project ExtLib: https://code.google.com/p/ocaml-extlib/

Makefile — a makefile that compiles the JME source files using ocamllex, ocamlyacc, and ocamlc into an executable binary

testall.sh — a shell script that executes all of the unit tests under the **tests** folder

tests folder — folder that contains all of the unit tests for the JME project

Chapter 6

Testing

The features of the language will be developed from end to end before moving on to the next feature. This means that the parser, abstract syntax tree, compiler and executable bytecode are all developed to support the current feature.

6.1 Sample Unit Tests

test-arith-mix1.jme

```
main() {
a = 2;
b = 17.6;
print(a + b);
print(a - b);
print(a * b);
print(b / a);
}
```

output:

```
19.6
-15.6
35.2
8.8
```

test-checktype1.jme

```
main()
{
 int = 8;
 float = 8.8;
 str = "String";
 bool = true;
 vec = [1,2,3];
 matx = [1,2; 3,4];
```

```
 map = new | "a" => "a", "b" => "b", "c" => "c" |;

 print("Check Int");
 print(is_Int(int));
 print(is_Int(float));

 print("Check Float");
 print(is_Float(float));
 print(is_Float(int));

 print("Check String");
 print(is_String(str));
 print(is_String(int));

 print("Check Boolean");
 print(is_Bool(bool));
 print(is_Bool(str));

 print("Check Vector");
 print(is_Vector(vec));
 print(is_Vector(bool));

 print("Check Matrix");
 print(is_Matrix(matx));
 print(is_Matrix(vec));

 print("Check Map");
 print(is_Map(map));
 print(is_Map(matx));

}
```

output:

```
Check Int
true
false
Check Float
true
false
Check String
true
false
Check Boolean
true
false
Check Vector
true
```

```
false
Check Matrix
true
false
Check Map
true
false
```

6.2 Unit Test Creation

One or more unit tests were created to test each feature in full. Each unit test is a small JME program designed to test the current feature. In theory each unit test would test all cases corresponding to a language feature. For some of the more complex features, like data structures, multiple unit tests were created to test individual aspects of that feature. A corresponding output file is also created with the expected output. The unit tests were saved under a tests folder and committed as part of the source control to Github.

6.3 Automated Testing

A shell script **testall.sh** was created to quickly test all unit tests. Two files are saved, the first one with a **.jme** extension is the unit test and the second one is a **.out** with the same name and contains the expected output. The shell script runs each unit test beneath the tests folder and compares the output of the unit test with the expected output. The shell script prints out whether the tests passed and if not generates a **.diff** file to illustrate the differences.

A shell script, **testall.sh**, will run all of the unit tests and compare the output to the expected output. Only if **testall.sh** runs successfully will the current feature be committed to the master branch.

Chapter 7

Lessons Learned

Several lessons were learned over the course of this project. The first being is **start early!**. This begins with starting early on Homework 1. The Homework 1 is a great exercise because it forces you to learn Ocaml, however if you start late on this assignment and are pressed with a deadline you might end up resorting to guess/check and not fully understand what you are doing. 99 problems on the Ocaml website is also a great exercise to help with the learning curve of Ocaml: http://ocaml.org/learn/tutorials/99problems.html. Commit early and often. Since you have to change about 5 files to run a new language feature(scanner, parser, ast, compiler...) it is important to have a detailed history in case you need to revert changes. Also be diligent with unit testing, when modifying the parser or abstract syntax tree it is easy to create side effects and break previous language features.

Appendix A

Built in Functions

The following is a full list of built in functions of the JME language.

print(*expr* **)** — outputs the literal value of the expression to the screen.

width(*expr* **)** — outputs an integer value of the number of columns of either a vector or matrix

height(*expr* **)** — outputs an integer value of the number of rows of a matrix. If the expression evaluates to a vector an integer value of 1 is always returned.

sqrt(*expr* **)** — returns the square root of an integer or float. The result will always be of type float.

sort(*vector* **)** — Expects a vector type. Will sort the vector elements in increasing order. The function does not return anything, it modifies the passed in vector. The sorting algorithm is very basic, if the vector contains mixed datatypes the results may be inconsistent.

mhas(*map, string* **)** — The first argument is a Map and the second argument is a potential key. The function returns a boolean value whether the given key is a key that exists within the given map.

is_Int(*expr* **)** — returns true if the given expression is an Integer type, false otherwise.

is_Bool(*expr* **)** — returns true if the given expression is a Boolean type, false otherwise.

is_String(*expr* **)** — returns true if the given expression is a String type, false otherwise.

is_Float(*expr* **)** — returns true if the given expression is a Float type, false otherwise.

is_Vector(*expr* **)** — returns true if the given expression is a Vector type, false otherwise.

is_Matrix(*expr* **)** — returns true if the given expression is a Matrix type, false otherwise.

is_Map(*expr* **)** — returns true if the given expression is a Map type, false otherwise.

Appendix B

Standard Library

The following is a full list of Standard Library functions in the first release of the JME language.

mean(*vector* **)** — Expects a vector type of integer and/or float elements. Returns the mean value of the elements of the vector.

median(*vector* **)** — Expects a vector type with all elements of the same type. Returns the median value of the elements of the vector.

mode(*vector* **)** — Expects a vector type. Returns the mode or most common value of the elements of the vector.

stdev(*vector* **)** — Expects a vector type of integer and/or float elements. Returns the standard deviation of the elements of the vector.

transpose(*expr* **)** — Expects a vector or matrix type. Returns the transpose of the passed in matrix by reflecting the elements over its main diagonal. If a vector is passed in a column matrix is returned.

Appendix C

Code Listing

C.1 ast.ml

```
type op = Add | Sub | Mult | Div | Exponent | Equal | Neq | Less
    | Leq | Greater | Geq | Mod


type expr =
    Literal of int
  | Float of float
  | Boolean of bool
  | String of string
  | Id of string
  | Binop of expr * op * expr
  | Assign of string * expr
  | Call of string * expr list
  | VectAssign of string * expr * expr
  | VectRef of string * expr
  | VectorInit of expr
  | Matrix of expr list list
  | MatrixInit of expr * expr
  | MatxRef of string * expr * expr
  | MatxAssign of string * expr * expr * expr
  | JMap of (expr * expr ) list
  | Noexpr

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt


type func_decl = {
    fname : string;
    formals : string list;
    locals : string list;
```

```
    body : stmt list;
  }

type program = string list * func_decl list

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | Float(f) -> string_of_float f
  | Boolean(b) -> string_of_bool b
  | String(s) -> s
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
  Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Mod -> "%"
      | Exponent -> "^"
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">="
    ) ^ " " ^
      string_of_expr e2
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el)
    ^ ")"
  | Matrix (m) -> "\n[" ^ (String.concat ";\n " (List.map (fun
    lexpr -> "" ^ (String.concat "," (List.map (fun e ->
    string_of_expr e) lexpr )) ^ "" ) m))
                ^ "]"
  | JMap (m) -> "|" ^ String.concat ", " (List.map (fun (k,v) ->
     string_of_expr k ^ "=>" ^ string_of_expr v) m) ^ "|"
  | VectorInit(e) -> "new " ^ "[" ^ (string_of_expr e) ^ "]"
  | MatrixInit(x,y) -> "new " ^ "[" ^ (string_of_expr x) ^ "] ["
     ^ (string_of_expr y) ^ "]"
  | MatxRef(v, x, y) -> v ^ "[" ^ string_of_expr x ^ "][" ^
    string_of_expr y ^ "]"
  | VectRef(v, e) -> v ^ "[" ^ string_of_expr e ^ "]"
  | VectAssign(v, e1, e2) -> v ^ "[" ^ string_of_expr e1 ^ "] =
    " ^ string_of_expr e2
  | MatxAssign(m, e1, e2, e3) -> m ^ "[" ^ string_of_expr e1 ^ "
    ][" ^ string_of_expr e2 ^ "] = " ^ string_of_expr e3
  | Noexpr -> ""

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^
     "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
```

```
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s

let string_of_vdecl id = "var " ^ id ^ ";\n"

let string_of_fdecl fdecl =
  fdecl.fname ^ "(" ^ String.concat ", " fdecl.formals ^ ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

C.2 bytecode.ml

```

type bstmt =
    Lit of Datatypes.dtypes (* Push a literal *)
  | Drp           (* Discard a value *)
  | Bin of Ast.op (* Perform arithmetic on top of stack *)
  | Lod of int    (* Fetch global variable *)
  | Str of int    (* Store global variable *)
  | Lfp of int    (* Load frame pointer relative *)
  | Sfp of int    (* Store frame pointer relative *)
  | Jsr of int    (* Call function by absolute address *)
  | Ent of int    (* Push FP, FP -> SP, SP += i *)
  | Rts of int    (* Restore FP, SP, consume formals, push result *)
  | Beq of int    (* Branch relative if top-of-stack is zero *)
  | Bne of int    (* Branch relative if top-of-stack is non-zero *)
  | Bra of int    (* Branch relative *)
  | Vec of int    (* vector init *)
```

```
  | Mat of int * int (* matrix init *)
  | Map of int     (* map init *)
  | Lodv of int    (* Load element of vector global *)
  | Lfpv of int    (* Load element of vector local *)
  | Lodm of int    (* Load element of matrix global *)
  | Lfpm of int    (* Load element of matrix local *)
  | Ulvec of int   (* Update element of vector local *)
  | Ugvec of int   (* Update element of vector global *)
  | Ulmat of int   (* Update element of matrix local *)
  | Ugmat of int   (* Update element of matrix global *)
  | Veci           (* create an empty vector and push on to stack
    *)
  | Mati           (* create an empty matrix and push on to stack
    *)
  | Hlt            (* Terminate *)

type prog = {
    num_globals : int;   (* Number of global variables *)
    text : bstmt array; (* Code for all the functions *)
  }


let string_of_stmt = function
    Lit(i) -> "Lit " ^ Datatypes.string_of_expr i
  | Drp -> "Drp"
  | Bin(Ast.Add) -> "Add"
  | Bin(Ast.Sub) -> "Sub"
  | Bin(Ast.Mult) -> "Mul"
  | Bin(Ast.Div) -> "Div"
  | Bin(Ast.Exponent) -> "Exponent"
  | Bin(Ast.Equal) -> "Eql"
  | Bin(Ast.Neq) -> "Neq"
  | Bin(Ast.Less) -> "Lt"
  | Bin(Ast.Leq) -> "Leq"
  | Bin(Ast.Greater) -> "Gt"
  | Bin(Ast.Geq) -> "Geq"
  | Bin(Ast.Mod) -> "Mod"
  | Lod(i) -> "Lod " ^ string_of_int i
  | Str(i) -> "Str " ^ string_of_int i
  | Lfp(i) -> "Lfp " ^ string_of_int i
  | Sfp(i) -> "Sfp " ^ string_of_int i
  | Jsr(i) -> "Jsr " ^ string_of_int i
  | Ent(i) -> "Ent " ^ string_of_int i
  | Rts(i) -> "Rts " ^ string_of_int i
  | Bne(i) -> "Bne " ^ string_of_int i
  | Beq(i) -> "Beq " ^ string_of_int i
  | Bra(i) -> "Bra " ^ string_of_int i
  | Vec(i) -> "Vect " ^ string_of_int i
  | Mat(i,e) -> "Mat " ^ string_of_int i ^ " " ^ string_of_int e
```

```
  | Map(i) -> "Map " ^ string_of_int i
  | Lodv(i) -> "Lodv " ^ string_of_int i
  | Lfpv(i) -> "Lfpv " ^ string_of_int i
  | Lodm(i) -> "Lodm " ^ string_of_int i
  | Lfpm(i) -> "Lfpm " ^ string_of_int i
  | Ulvec(i) -> "Ulvec " ^ string_of_int i
  | Ugvec(i) -> "Ugvec " ^ string_of_int i
  | Ulmat(i) -> "Ulmat " ^ string_of_int i
  | Ugmat(i) -> "Ugmat " ^ string_of_int i
  | Veci -> "Veci"
  | Mati -> "Mati"
  | Hlt     -> "Hlt"

let string_of_prog p =
  string_of_int p.num_globals ^ " global variables\n" ^
  let funca = Array.mapi
      (fun i s -> string_of_int i ^ " " ^ string_of_stmt s) p.
    text
  in String.concat "\n" (Array.to_list funca)
```

C.3 compile.ml

```
open Ast
open Bytecode

module StringMap = Map.Make(String)

(* Symbol table: Information about all the names in scope *)
type env = {
    function_index : int StringMap.t; (* Index for each function
     *)
    global_index   : int StringMap.t; (* "Address" for global
    variables *)
    mutable local_index    : int StringMap.t; (* FP offset for
    args, locals *)
  }

(* val enum : int -> 'a list -> (int * 'a) list *)
let rec enum stride n = function
    [] -> []
  | hd::tl -> (n, hd) :: enum stride (n+stride) tl

(* val string_map_pairs StringMap 'a -> (int * 'a) list ->
    StringMap 'a  *)
let string_map_pairs map pairs =
```

```
  List.fold_left (fun m (i, n) -> StringMap.add n i m) map pairs

(*debug to view local variables *)
let print_locals s i =
    print_string (s ^ "->"); print_int i; print_endline "";;

(** Translate a program in AST form into a bytecode program.
   Throw an
    exception if something is wrong, e.g., a reference to an
   unknown
    variable or function *)
let translate (globals, functions) =

  (* Allocate "addresses" for each global variable *)
  let global_indexes = string_map_pairs StringMap.empty (enum 1
   0 globals) in

  (* Assign indexes to function names; built-in "print" is
   special *)
  let built_in_functions = StringMap.add "print" (-1) StringMap.
   empty in
  let built_in_functions = StringMap.add "width" (-2)
   built_in_functions in
  let built_in_functions = StringMap.add "height" (-3)
   built_in_functions in
  let built_in_functions = StringMap.add "sqrt" (-4)
   built_in_functions in
  let built_in_functions = StringMap.add "sort" (-5)
   built_in_functions in
  let built_in_functions = StringMap.add "mhas" (-6)
   built_in_functions in
  let built_in_functions = StringMap.add "is_Int" (-7)
   built_in_functions in
  let built_in_functions = StringMap.add "is_Float" (-8)
   built_in_functions in
  let built_in_functions = StringMap.add "is_Bool" (-9)
   built_in_functions in
  let built_in_functions = StringMap.add "is_String" (-10)
   built_in_functions in
  let built_in_functions = StringMap.add "is_Vector" (-11)
   built_in_functions in
  let built_in_functions = StringMap.add "is_Matrix" (-12)
   built_in_functions in
  let built_in_functions = StringMap.add "is_Map" (-13)
   built_in_functions in
  let function_indexes = string_map_pairs built_in_functions
      (enum 1 1 (List.map (fun f -> f.fname) functions)) in
```

```
(* Translate a function in AST form into a list of bytecode
  statements *)
let translate env fdecl =
  (* Bookkeeping: FP offsets for locals and arguments *)
  let num_formals = List.length fdecl.formals
  and local_offsets = enum 1 1 fdecl.locals
  and formal_offsets = enum (-1) (-2) fdecl.formals in
  let env = { env with local_index = string_map_pairs
    StringMap.empty (local_offsets @ formal_offsets) } in

  let rec expr = function
Literal i -> [Lit (Datatypes.Int i)]
    | Float f -> [Lit (Datatypes.Float f)]
    | Boolean b -> [Lit (Datatypes.Boolean b)]
    | String s -> [Lit (Datatypes.String (Datatypes.
  stripQuotes s))]
    | Id s -> (*print_string "numformals: "; print_int
  num_formals; print_endline "";
              print_int (StringMap.cardinal env.local_index);
  print_endline ""; StringMap.iter print_locals env.local_index
  ; *)
  (try [Lfp (StringMap.find s env.local_index)]
       with Not_found -> try [Lod (StringMap.find s env.
  global_index)]
       with Not_found -> raise (Failure ("undeclared variable
   " ^ s)))
    | Binop (e1, op, e2) -> expr e1 @ expr e2 @ [Bin op]
    | Assign (s, e) -> expr e @
  (try [Sfp (StringMap.find s env.local_index)]
    with Not_found -> try [Str (StringMap.find s env.
  global_index)]
  with Not_found -> let lclindex = StringMap.cardinal env.
  local_index - num_formals + 1 in
                     ignore(env.local_index <- StringMap.add
  s lclindex env.local_index);
                     [Sfp lclindex]
          (*raise (Failure ("undeclared variable " ^ s))*))
    | VectRef (s, e) -> expr e @
       (try [Lfpv (StringMap.find s env.local_index)]
           with Not_found -> try [Lodv (StringMap.find s env.
  global_index)]
         with Not_found -> raise (Failure ("undeclared
  variable " ^ s)))
    | VectAssign (s, e1, e2) -> expr e2 @ expr e1 @
       (try [Ulvec (StringMap.find s env.local_index)]
           with Not_found -> try [Ugvec (StringMap.find s env.
  global_index)]
         with Not_found -> raise (Failure ("undeclared
  variable " ^ s)))
```

```
      | Call (fname, actuals) -> (try
    (List.concat (List.map expr (List.rev actuals))) @
    [Jsr (StringMap.find fname env.function_index) ]
        with Not_found -> raise (Failure ("undefined function "
    ^ fname)))
      | Matrix m -> if List.length m = 1 then let vect = List.hd
    m in
                      (List.concat (List.map expr vect)) @ [
    Vec (List.length vect) ]
                  else
                    let dimx = List.length m in
                    let dimy = List.length (List.hd m) in
                    ignore(Util.checkmatrix m); (List.concat (
    List.map expr (List.concat m))) @ [Mat (dimx,dimy) ]
      | JMap m -> List.concat (List.map (fun (k,v) -> (expr v) @
    (expr k) ) m) @ [Map ( List.length m) ]
      | VectorInit e -> expr e @ [Veci]
      | MatrixInit (x,y) -> expr x @ expr y @ [Mati]
      | MatxRef (s, x, y) -> expr x @ expr y @
          (try [Lfpm (StringMap.find s env.local_index)]
           with Not_found -> try [Lodm (StringMap.find s env.
    global_index)]
         with Not_found -> raise (Failure ("undeclared
    variable " ^ s)))
      | MatxAssign (s, e1, e2, e3) -> expr e3 @ expr e1 @ expr
    e2 @
        (try [Ulmat (StringMap.find s env.local_index)]
           with Not_found -> try [Ugmat (StringMap.find s env.
    global_index)]
         with Not_found -> raise (Failure ("undeclared
    variable " ^ s)))
      | Noexpr -> []

    in let rec stmt = function
  Block sl     ->  List.concat (List.map stmt sl)
      | Expr e        -> expr e @ [Drp]
      | Return e      -> expr e @ [Rts num_formals]
      | If (p, t, f) -> let t' = stmt t and f' = stmt f in
  expr p @ [Beq(2 + List.length t')] @
  t' @ [Bra(1 + List.length f')] @ f'
      | For (e1, e2, e3, b) ->
    stmt (Block([Expr(e1); While(e2, Block([b; Expr(e3)]))]))
      | While (e, b) ->
    let b' = stmt b and e' = expr e in
    [Bra (1+ List.length b')] @ b' @ e' @
    [Bne (-(List.length b' + List.length e'))]

    in [Ent (StringMap.cardinal env.local_index - num_formals)]
   @      (* Entry: allocate space for locals *)
```

```
      stmt (Block fdecl.body) @  (* Body *)
      [Lit (Datatypes.Int 0); Rts num_formals]   (* Default = return 0 *)

  in let env = { function_index = function_indexes;
      global_index = global_indexes;
      local_index = StringMap.empty } in

  (* Code executed to start the program: Jsr main; halt *)
  let entry_function = try
    [Jsr (StringMap.find "main" function_indexes); Hlt]
  with Not_found -> raise (Failure ("no \"main\" function"))
  in

  (* Compile the functions *)
  let func_bodies = entry_function :: List.map (translate env) functions in

  (* Calculate function entry points by adding their lengths *)
  let (fun_offset_list, _) = List.fold_left
      (fun (l,i) f -> (i :: l, (i + List.length f))) ([],0) func_bodies in
  let func_offset = Array.of_list (List.rev fun_offset_list) in

  { num_globals = List.length globals;
    (* Concatenate the compiled functions and replace the function
       indexes in Jsr statements with PC values *)
    text = Array.of_list (List.map (function
  Jsr i when i > 0 -> Jsr func_offset.(i)
      | _ as s -> s) (List.concat func_bodies))
  }
```

C.4 datatypes.ml

```
(* Data types of language *)
module StringMap = Map.Make(String)

type dtypes =
      Null
    | Int of int
    | Float of float
    | Boolean of bool
    | String of string
    | Vector of dtypes array
```

```
    | Matrix of dtypes array array
    | JMap of dtypes StringMap.t


let string_of_dtype = function
    Int(s) -> "Int "
    | Float(s) -> "Float"
    | Boolean(s) -> "Boolean"
    | String(s) -> "String"
    | Vector(s) -> "Vector"
    | Matrix(s) -> "Matrix"
    | JMap(s) -> "Map"
    | Null -> "Null"

let rec string_of_expr = function
    Int(s) -> string_of_int s
    | Float(s) -> string_of_float s
    | Boolean(s) -> string_of_bool s
    | String(s) -> s
    | Vector(s) -> "[" ^ String.concat "," (List.map
    string_of_expr (Array.to_list s)) ^ "]"
    | Matrix(s) ->  "[" ^ (String.concat ";\n " (List.map (fun
    lexpr -> "" ^
                    (String.concat "," (List.map (fun e ->
    string_of_expr e) (Array.to_list lexpr) ))  ^ "" ) (Array.
    to_list s)))
                ^ "]"
    | JMap(s) -> let string_pairs k v str= (k ^ "=>" ^
                    string_of_expr v ^ ", " ^ str) in
                "|" ^ (StringMap.fold string_pairs s "") ^ "|"
    | Null -> "null"

let getVectElement i v =
    match i, v with
    Int(i), Vector(v) -> v.(i)
    | String(s), JMap(m) -> StringMap.find s m
    |_ -> raise(Failure("invalid vector access"))

let keyExists i v =
    match i, v with
     String(s), JMap(m) -> if StringMap.mem s m then Boolean(
    true) else Boolean(false)
    |_ -> raise(Failure("invalid map access"))

let updateVectElement i v nv =
    match i, v with
    Int(i), Vector(v) -> v.(i) <- nv
    |_ -> raise(Failure("Invalid update structure"))

```

```
let printstack stack =
    for i = 0 to (Array.length stack -1) do
        print_endline ((string_of_dtype stack.(i)) ^ " " ^ (
    string_of_expr stack.(i)))
    done

let add a b =
    match a, b with
    Int(a), Int(b) -> Int (a + b)
    | Float(a), Float(b) -> Float (a +. b)
    | Int(a), Float(b) -> Float ((float_of_int a) +. b)
    | Float(a), Int(b) -> Float (a +. (float_of_int b))
    | String(a), String(b) -> String (a ^ b)
    |_ -> raise(Failure("invalid addition"))

let subtract a b =
    match a, b with
    Int(a), Int(b) -> Int (a - b)
    | Float(a), Float(b) -> Float (a -. b)
    | Int(a), Float(b) -> Float ((float_of_int a) -. b)
    | Float(a), Int(b) -> Float (a -. (float_of_int b))
    |_ -> raise(Failure("invalid subtraction"))

let multiply a b =
    match a, b with
    Int(a), Int(b) -> Int (a * b)
    | Float(a), Float(b) -> Float (a *. b)
    | Int(a), Float(b) -> Float ((float_of_int a) *. b)
    | Float(a), Int(b) -> Float (a *. (float_of_int b))
    |_ -> raise(Failure("invalid multiplication"))

let divide a b =
    match a, b with
    Int(a), Int(b) -> Int (a / b)
    | Float(a), Float(b) -> Float (a /. b)
    | Int(a), Float(b) -> Float ((float_of_int a) /. b)
    | Float(a), Int(b) -> Float (a /. (float_of_int b))
    |_ -> raise(Failure("invalid division"))

let power a b =
    match a, b with
    Int(a), Int(b) -> (* if exponentiation by integer implicitly
     convert to float *)
                        Float((float_of_int a) ** (float_of_int b)
    )
    | Float(a), Float(b) -> Float (a ** b)
    | Int(a), Float(b) -> Float ((float_of_int a) ** b)
    | Float(a), Int(b) -> Float (a ** (float_of_int b))
    |_ -> raise(Failure("invalid arguments for exponents"))
```

```
let remainder a b =
    match a, b with
    Int(a), Int(b) -> Int(a mod b)
    | Float(a), Float(b) -> Float(mod_float a b)
    |_ -> raise(Failure("invalid arguments for remainder"))

let equal a b =
    match a, b with
    Int(a), Int(b) -> Boolean(a = b)
    | Float(a), Float(b) -> Boolean (a = b)
    | Boolean(a), Boolean(b) -> Boolean (a = b)
    |_ -> Boolean(false)

let nequal a b =
    match a, b with
    Int(a), Int(b) -> Boolean(a != b)
    | Float(a), Float(b) -> Boolean (a != b)
    | Boolean(a), Boolean(b) -> Boolean (a != b)
    |_ -> Boolean(false)

let lessthn a b =
    match a, b with
    Int(a), Int(b) -> Boolean(a < b)
    | Float(a), Float(b) -> Boolean (a < b)
    |_ -> Boolean(false)

let lessthneq a b =
    match a, b with
    Int(a), Int(b) -> Boolean(a <= b)
    | Float(a), Float(b) -> Boolean (a <= b)
    |_ -> Boolean(false)

let greatthn a b =
    match a, b with
    Int(a), Int(b) -> Boolean(a > b)
    | Float(a), Float(b) -> Boolean (a > b)
    |_ -> Boolean(false)

let greatthneq a b =
    match a, b with
    Int(a), Int(b) -> Boolean(a >= b)
    | Float(a), Float(b) -> Boolean (a >= b)
    |_ -> Boolean(false)

let stripQuotes str =
    String.sub str 1 ((String.length str) -2)

let to_Float i =
```

```
    match i with
    Int i -> float_of_int i
    | Float i -> i
    |_ -> raise(Failure ("Expected Float or Integer got " ^
    string_of_dtype i ^ " " ^ string_of_expr i))

let to_Bool i =
    match i with
    Boolean i -> i
    |_ -> raise(Failure ("Expected Boolean got " ^
    string_of_dtype i ^ " " ^ string_of_expr i))

let to_Vector i =
    match i with
    Vector i -> i
    |_ -> raise(Failure ("Expected Vector got " ^
    string_of_dtype i ^ " " ^ string_of_expr i))

let to_JMap m =
    match m with
    JMap m -> m
     |_ -> raise(Failure ("Expected Map got " ^ string_of_dtype
    m ^ " " ^ string_of_expr m))

let to_Matrix i =
    match i with
    Matrix i -> i
    |_ -> raise(Failure ("Expected Matrix got " ^
    string_of_dtype i ^ " " ^ string_of_expr i))

let to_Map m =
    match m with
    JMap m -> m
    |_ -> raise(Failure ("Expected Map got " ^ string_of_dtype m
     ^ " " ^ string_of_expr m))

let to_Int i=
    match i with
    Int i  -> i
    | _     -> raise(Failure ("Expected Integer got " ^
    string_of_dtype i ^ " " ^ string_of_expr i))

let to_String s =
    match s with
    String s -> s
    | _ -> raise(Failure ("Expected String got " ^
    string_of_dtype s ^ " " ^ string_of_expr s))

let is_Int i =
```

```
    match i with
    Int i -> true
    |_ -> false

let is_Float f =
    match f with
    Float f -> true
    |_ -> false

let is_String s =
    match s with
    String s -> true
    |_ -> false

let is_Bool s =
    match s with
    Boolean s -> true
    |_ -> false

let is_Vector s =
    match s with
    Vector s -> true
    |_ -> false

let is_Matrix s =
    match s with
    Matrix s -> true
    |_ -> false

let is_JMap s =
    match s with
    JMap s -> true
    |_ -> false
```

C.5 execute.ml

```
open Ast
open Bytecode
open Datatypes
open Util

(* Stack layout just after "Ent":

              <--- SP
   Local n
```

```
    ...
    Local 0
    Saved FP    <—— FP
    Saved PC
    Arg 0
    ...

    Arg n *)

module StringMap = Map.Make(String)

let execute_prog prog =
  let stack = Array.make 1024 Null
  and globals = Array.make prog.num_globals Null in


  let fillMatrix x y top =
    let m = Array.make_matrix x y Null in
        for i = 0 to x -1 do
            for j = 0 to y -1 do
                m.(i).(j) <- stack.(top + (i*(y) +j))
            done;
        done;
        m
  in

  let fillVector n top =
    if n == 0 then Array.make 0 Null
        else let v = Array.make n Null in
            for i = 0 to n - 1 do
                v.(i) <- stack.(top - ((n-1) - i))
            done;
            v in
  let rec fillMap t n map =
    if n == t then map
        else
            fillMap (t+2) (n) (StringMap.add (to_String stack.(t
     +1)) stack.(t) map) in

  let rec exec fp sp pc = match prog.text.(pc) with
    Lit i  -> stack.(sp) <- i; exec fp (sp+1) (pc+1)
  | Drp    -> exec fp (sp-1) (pc+1)
  | Bin op -> let op1 = stack.(sp-2) and op2 = stack.(sp-1) in
      stack.(sp-2) <- (
      match op with
  Add      -> add op1 op2
      | Sub      -> subtract op1 op2
      | Mult     -> multiply op1 op2
      | Div      -> divide op1 op2
```

```
      | Exponent -> power op1 op2
      | Equal   -> equal op1 op2
      | Neq     -> nequal op1 op2
      | Less    -> lessthn op1 op2
      | Leq     -> lessthneq op1 op2
      | Greater -> greatthn op1 op2
      | Geq     -> greatthneq op1 op2
      | Mod     -> remainder op1 op2) ;
     exec fp (sp-1) (pc+1)
  | Lod i   -> stack.(sp)   <- globals.(i)  ; exec fp (sp+1) (pc+1)
  | Str i   -> globals.(i)  <- stack.(sp-1) ; exec fp sp     (pc+1)
  | Lfp i   -> stack.(sp)   <- stack.(fp+i) ; exec fp (sp+1) (pc+1)
  | Sfp i   -> stack.(fp+i) <- stack.(sp-1) ; exec fp sp     (pc+1)
  | Jsr(-1) -> print_endline (string_of_expr stack.(sp-1)) ; exec fp sp (pc+1)
  | Jsr(-2) -> stack.(sp-1) <- getWidth stack.(sp-1); exec fp sp (pc+1)
  | Jsr(-3) -> stack.(sp-1) <- getHeight stack.(sp-1); exec fp sp (pc+1)
  | Jsr(-4) -> stack.(sp-1) <- Float (sqrt (to_Float stack.(sp-1))); exec fp sp (pc+1)
  | Jsr(-5) -> sortVect stack.(sp-1); exec fp sp (pc+1)
  | Jsr(-6) -> stack.(sp-1) <- keyExists stack.(sp-2) stack.(sp-1); exec fp sp (pc+1)
  | Jsr(-7) -> stack.(sp-1) <- Boolean( is_Int stack.(sp-1)); exec fp sp (pc+1)
  | Jsr(-8) -> stack.(sp-1) <- Boolean( is_Float stack.(sp-1)); exec fp sp (pc+1)
  | Jsr(-9) -> stack.(sp-1) <- Boolean( is_Bool stack.(sp-1)); exec fp sp (pc+1)
  | Jsr(-10) -> stack.(sp-1) <- Boolean( is_String stack.(sp-1)) ; exec fp sp (pc+1)
  | Jsr(-11) -> stack.(sp-1) <- Boolean( is_Vector stack.(sp-1)) ; exec fp sp (pc+1)
  | Jsr(-12) -> stack.(sp-1) <- Boolean( is_Matrix stack.(sp-1)) ; exec fp sp (pc+1)
  | Jsr(-13) -> stack.(sp-1) <- Boolean( is_JMap stack.(sp-1)); exec fp sp (pc+1)
  | Jsr i   -> stack.(sp)   <- Int (pc + 1)       ; exec fp (sp+1) i
  | Ent i   -> stack.(sp)   <- Int fp             ; exec sp (sp+i+1) (pc+1)
  | Rts i   -> let new_fp = to_Int stack.(fp) and new_pc = to_Int stack.(fp-1) in
```

```
            stack.(fp-i-1) <- stack.(sp-1) ; exec new_fp (fp-i) new_pc
  | Beq i   -> exec fp (sp-1) (pc + if to_Bool(equal stack.(sp-1) (Boolean false)) then i else 1)
  | Bne i   -> exec fp (sp-1) (pc + if to_Bool(nequal stack.(sp-1) (Boolean false)) then i else 1)
  | Bra i   -> exec fp sp (pc+i)
  | Vec i -> stack.(sp-i) <- Vector(fillVector i (sp -1)); exec fp (sp-i +1) (pc+1)
  | Mat (x,y) -> stack.(sp - (x*y)) <- Matrix( fillMatrix x y (sp - x*y)); exec fp (sp - x*y +1) (pc+1)
  | Veci -> stack.(sp-1) <- Vector(Array.make (to_Int(stack.(sp-1))) Null); exec fp sp (pc+1)
  | Mati -> stack.(sp-2) <- Matrix(Array.make_matrix (to_Int(stack.(sp-2))) (to_Int(stack.(sp-1))) Null);exec fp (sp-1) (pc+1)
  | Map i -> stack.(sp - 2*i) <- JMap(fillMap (sp-2*i ) sp StringMap.empty); exec fp (sp-2*i+1) (pc+1)
  | Lodv i -> let nth = stack.(sp-1) in
              stack.(sp-1) <- getVectElement nth globals.(i); exec fp (sp) (pc+1)
  | Lfpv i -> let nth = stack.(sp-1) in
              stack.(sp-1) <- getVectElement nth stack.(fp+i); exec fp (sp) (pc+1)
  | Lfpm i -> let x = stack.(sp-2) in
            let y = stack.(sp-1) in
              stack.(sp-2) <- getMatxElement x y stack.(fp+i); exec fp (sp -1) (pc+1)
  | Lodm i -> let x = stack.(sp-2) in
            let y = stack.(sp-1) in
              stack.(sp-2) <- getMatxElement x y globals.(i); exec fp (sp -1) (pc+1)
  | Ulvec i -> if is_Int (stack.(sp-1)) then  updateVectElement stack.(sp-1) stack.(fp+i) stack.(sp-2)
                  else  stack.(fp+i) <- JMap((StringMap.add (to_String(stack.(sp-1))) stack.(sp-2) (to_JMap(stack.(fp+i)))));
               exec fp sp (pc+1)
  | Ugvec i -> if is_Int (stack.(sp-1)) then  updateVectElement stack.(sp-1) globals.(i) stack.(sp-2)
                  else  globals.(i) <- JMap((StringMap.add (to_String(stack.(sp-1))) stack.(sp-2) (to_JMap(globals.(i)))));


              (*let indx = stack.(sp-1) and nval = stack.(sp-2) and svec = (to_Vector globals.(i)) in
              svec.(to_Int indx) <- nval;*)
               exec fp sp (pc+1)
```

```
  | Ulmat i -> let x = stack.(sp-2) and y = stack.(sp-1) and
    nval = stack.(sp-3) and svec = (to_Matrix stack.(fp+i)) in
                 svec.(to_Int x).(to_Int y) <- nval;
                  exec fp sp (pc+1)
  | Ugmat i -> let indx = stack.(sp-1) and nval = stack.(sp-2)
    and svec = (to_Vector globals.(i)) in
                 svec.(to_Int indx) <- nval;
                  exec fp sp (pc+1)
  | Hlt      -> ()

  in exec 0 0 0
```

C.6 jme.ml

```
type action = Ast | Bytecode | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);
            ("-b", Bytecode);
            ("-c", Compile)]
  else Compile in

  (*let stdlib = input_all (open_in "stdlib.jme") in
  let currprog = input_all stdin in
  let fullprog = Lexing.from_string (stdlib ^ currprog) in *)
  let program = if Array.length Sys.argv = 3 && Sys.argv.(2) = "
    -wo" then (Parser.program Scanner.token (Lexing.from_channel
    stdin))
                else (Parser.program Scanner.token (Lexing.
    from_string ((Util.input_all (open_in "stdlib.jme")) ^ (Util.
    input_all stdin)))) in
  match action with
    Ast -> let listing = Ast.string_of_program program
           in print_string listing
  | Bytecode -> let listing =
      Bytecode.string_of_prog (Compile.translate program)
    in print_endline listing
  | Compile -> Execute.execute_prog (Compile.translate program)
```

C.7 parser.mly

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA LBRACKET RBRACKET
%token NEW POINTER BAR
%token PLUS MINUS TIMES DIVIDE ASSIGN MOD EXPONENT
%token EQ NEQ LT LEQ GT GEQ
%token RETURN IF ELSE FOR WHILE VAR
%token <int> LITERAL
%token <float> FLOAT
%token <bool> BOOLEAN
%token <string> STRING
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%left EXPONENT

%start program
%type <Ast.program> program

%%

program:
    /* nothing */ { [], [] }
  | program vdecl { ($2 :: fst $1), snd $1 }
  | program fdecl { fst $1, ($2 :: snd $1) }

fdecl:
    ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
      { { fname = $1;
    formals = $3;
    locals = [];
    body = List.rev $6 } }

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

formal_list:
     ID                    { [$1] }
```

```
  | formal_list COMMA ID { $3 :: $1 }

/*
vdecl_list:
        { [] }
  | vdecl_list vdecl { $2 :: $1 }*/


vdecl:
   VAR ID SEMI { $2 }

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    expr SEMI { Expr($1) }
  | RETURN expr SEMI { Return($2) }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block
    ([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt
     { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }


expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }

expr:
    LITERAL          { Literal($1) }
  | FLOAT            { Float($1) }
  | BOOLEAN          { Boolean($1) }
  | STRING           { String($1) }
  | ID               { Id($1) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr EXPONENT expr { Binop($1, Exponent, $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater,  $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr MOD    expr { Binop($1, Mod, $3) }
```

```
  | ID ASSIGN expr    { Assign($1, $3) }
  | NEW LBRACKET expr RBRACKET {VectorInit($3) }
  | NEW LBRACKET expr RBRACKET LBRACKET expr RBRACKET { MatrixInit($3,$6) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | ID LBRACKET expr RBRACKET { VectRef($1,$3) }
  | ID LBRACKET expr RBRACKET LBRACKET expr RBRACKET { MatxRef($1, $3, $6) }
  | ID LBRACKET expr RBRACKET ASSIGN expr { VectAssign($1,$3, $6) }
  | ID LBRACKET expr RBRACKET LBRACKET expr RBRACKET ASSIGN expr { MatxAssign($1,$3, $6, $9) }
  | LBRACKET mat_opt RBRACKET {Matrix($2) }
  | NEW BAR keyvalue_opt BAR {JMap($3) }
  | LPAREN expr RPAREN { $2 }

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

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

   mrow:
   { [] }
  | expr { [$1] }
  | mrow COMMA expr { $3 :: $1 }

    mat_opt:
   mrow { [List.rev $1] }
  | mat_opt SEMI mrow { $1 @ [List.rev $3] }

     keyvalue_opt:
    { [] }
    | keyvalue_list { List.rev $1 }

    keyvalue_list:
    expr POINTER expr { [$1, $3] }
    | keyvalue_list COMMA expr POINTER expr {($3, $5)::$1 }
```

C.8 scanner.mll

```
{ open Parser }

```

```
rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"      { comment lexbuf }           (* Comments *)
| '('       { LPAREN }
| ')'       { RPAREN }
| '{'       { LBRACE }
| '}'       { RBRACE }
| '['       { LBRACKET }
| ']'       { RBRACKET }
| ';'       { SEMI }
| ','       { COMMA }
| '+'       { PLUS }
| '-'       { MINUS }
| '*'       { TIMES }
| '/'       { DIVIDE }
| '%'       { MOD }
| '^'       { EXPONENT }
| '='       { ASSIGN }
| "=="      { EQ }
| "!="      { NEQ }
| '<'       { LT }
| "<="      { LEQ }
| ">"       { GT }
| ">="      { GEQ }
| "if"      { IF }
| "else"    { ELSE }
| "for"     { FOR }
| "while"   { WHILE }
| "return"  { RETURN }
| "var"     { VAR }
| "new"     { NEW }
| "=>"      { POINTER }
| "|"       { BAR }
| "true" | "false" as boolean { BOOLEAN(bool_of_string boolean)}
| '"'[^ '\r' '\n' '\"']*'"' as str {STRING (str)}
| ['-']?['0'-'9']*['.']['0'-'9']+ as lxm { FLOAT(float_of_string lxm) }
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  "*/" { token lexbuf }
| _    { comment lexbuf }
```

C.9 stdlib.jme

```

mean(a)
{

  total = 0.0;
  for (i = 0 ; i < width(a) ; i = i + 1) {
    total = a[i] + total;
  }
  return total / width(a);
}

median(a)
{

 sort(a);
 total = width(a);
 if((total % 2) == 0) {
     num1 = a[total/2];
     num2 = a[(total/2) - 1];
     return (num1 + 0.0 + num2 + 0.0) /2;
    }
  else {
    return a[width(a)/2];
  }
}


mode(a)
{

  sort(a);

  curr = a[0];
  mode = a[0];
  count = 1;
  prevcount = 1;
  for (i = 1 ; i < width(a) ; i = i + 1) {
    if(curr == a[i]) {
        count = count +1;
    } else {
      if(count > prevcount) {
        prevcount = count;
        count = 1;
        mode = a[i-1];
      }
      curr = a[i];
```

```
    }
  }
  return mode;
}

stdev(a)
{

  newarr = new [width(a)];

    avg = mean(a);
    for(i = 0; i < width(a); i = i+1) {
        newarr[i] = (a[i] - avg) ^ 2;
    }

    avg = mean(newarr);

    return sqrt(avg);
}

transpose(mat) {

tran = new [width(mat)][height(mat)];

/* check if its actually a vector */
if(height(mat) == 1) {
    for(i = 0; i < width(mat); i = i +1) {
        tran[i][0] = mat[i];
    }
}
else {
for (i = 0 ; i < height(mat) ; i = i + 1) {
    for(j=0; j < width(mat); j = j + 1) {
        tran[j][i] = mat[i][j];
    }
}
}

return tran;

}
```

C.10 util.ml

```
open Datatypes
```

```

let checkmatrix m =
            let size = List.length (List.hd m) in
             List.iter (fun chksize ->
                         if List.length chksize != size
                         then raise (Failure ("unequal row sizes in matrix"))) m


let getMatxElement x y m =
    match x,y, m with
    Int(x), Int(y), Matrix(m) -> m.(x).(y)
    |_ -> raise(Failure("invalid matrix access"))


let getWidth s =
    match s with
    | Vector(s) -> Int (Array.length (s))
    | Matrix(s) -> Int (Array.length s.(0))
    | JMap(m) -> Int(StringMap.cardinal m)
    |_ -> raise(Failure(("width() cannot be applied to type " ^ Datatypes.string_of_dtype s ^ " " ^ Datatypes.string_of_expr s)))

let getHeight s =
    match s with
    | Vector(s) -> Int (1)
    | Matrix(s) -> Int (Array.length s)
    |_ -> raise(Failure(("width() cannot be applied to type " ^ Datatypes.string_of_dtype s ^ " " ^ Datatypes.string_of_expr s)))

let sortVect s =
    match s with
    | Vector(s) -> Array.sort compare s
    |_ -> raise(Failure(("width() cannot be applied to type " ^ Datatypes.string_of_dtype s ^ " " ^ Datatypes.string_of_expr s)))

let buf_len = 8192

let input_all ic =
  let rec loop acc total buf ofs =
    let n = input ic buf ofs (buf_len - ofs) in
    if n = 0 then
      let res = String.create total in
      let pos = total - ofs in
      let _ = String.blit buf 0 res pos ofs in
```

```
      let coll pos buf =
        let new_pos = pos - buf_len in
        String.blit buf 0 res new_pos buf_len;
        new_pos in
      let _ = List.fold_left coll pos acc in
      res
    else
      let new_ofs = ofs + n in
      let new_total = total + n in
      if new_ofs = buf_len then
        loop (buf :: acc) new_total (String.create buf_len) 0
      else loop acc new_total buf new_ofs in
  loop [] 0 (String.create buf_len) 0
```

Bibliography

[1] Dennis M. Ritchie, *C Reference Manual*. Bell Telephone Laboratories, Murray Hill, New Jersey 07974

[2] Christopher Conway, Cheng-Hong Li, Megan Pengelly *Pencil: A Petri Net Specification Language for Java*. Columbia University, New York, December 2002