JME Language Reference Manual

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 (such as mean, median, mode…) but will allow the user to express any number of statistical analysis algorithms through the creation of functions.

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

2 Lexical Conventions

A JME program is typically written in a single file with a .jme extension. The JME language contains a set of tokens that includes identifiers, keywords, constants, expression operators and separators. The compiler ignores Whitespace, 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.

2.1 Whitespace

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

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.

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.

letter ➔ **[a-z]|[A-Z]**
digit ➔ **[0-9]**
Identifier ➔ *letter*(*letter* | *digit*)*

2.4 Keywords

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

break	**else**	**false**
for	**function**	**if**
null	**return**	**then**
true	**while**	**has**
length	**height**	**in**
key		

2.5 Constants

There are several types of constants defined as follows:

2.5.1 Integer Constants

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

Integer ➔ *digit*+

2.5.2 Floating Constants

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. Either the fraction part or the integer part is optional but not both. Every floating constant is taken to be double precision.

Float ➔
digit+**.***digit**
| *digit**.*digit*+

2.5.2 String Constants

A String is a sequence of characters enclosed in double quotes. The empty string, containing no characters, can be defined as two double quotes with no space: "". To define a string that contains a double quote it must be escaped using the '\' character: "\"".

String ➔ "(*letter* | *digit*) * "

2.5.3 Boolean Constants

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".

Boolean ➔ **true** | **false**

2.6 Separators

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

{ } ( ) [ ] ; . ,

2.7 Operators

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

+ − * /	Plus, Minus, Multiplication, Division.
=	Assignment to an Identifier
==	Logical equals
!	Unary Logical not
!=	Not Equal
&&	Logical AND
||	Logical OR
=>	Assignment within a Map
> < >= <=	Logical: Greater than, Less than, Greater than or equal, Less than or Equal

3 Expressions and Operators

Expressions can use operators to define a mathematical or logical computation. Standard mathematical order of operations is respected when evaluating an expression. 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.

3.1 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) or a Boolean constant (true or false).

Expression ➔ (*constant* | *identifier*) **;**

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.

3.2.1 *expr* * *expr*

Will return the multiplication result of the two expressions.

3.2.2 *expr* / *expr*

Will return the division of the left expression by the right expression. If two integers are divided and have a non integer result a floating point number will be returned.

3.2.3 *expr* – *expr*

Will return the left expression subtracted by the right expression.

3.2.4 *expr* + *expr*

Will return the addition of the two expressions. This is the only binary operator that can be used with the String type. In the case of two String expressions the right String expression will be concatenated to the end of the left string expression.

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 with the exception of integer and floating point numbers.

3.3.1 *expr* == *expr*

This operator evaluates the equality of the two expressions.

3.3.2 *expr* != *expr*

This operator evaluates if the two expressions are different.

3.3.3 *expr* > *expr*

This operator evaluates if the left expression is greater than the right expression. Restricted to integer and floating point types.

3.3.4 *expr* >= *expr*

This operator evaluates if the left expression is greater than or equal to the right expression. Restricted to integer and floating point types.

3.3.5 *exp < expr*
This operator evaluates if the left expression is less than the right expression. Restricted to integer and floating point types.

3.3.6 *expr <= expr*
This operator evaluates if the left expression is less than or equal to the right expression. Restricted to integer and floating point types.

3.4 *!boolean-expr*
The unary not operator evaluates the Boolean expression and returns the opposite value.

3.5 Logical Operators
Logical operators can only be applied to Boolean expressions.

3.5.1 boolean-expr && boolean-expr
The logical AND operator evaluates to true if both the left Boolean expression and the right Boolean expression are true.

3.5.2 boolean-expr || boolean-expr
The logical OR operator evaluates to true if either the left Boolean expression or the right Boolean expression evaluate to true.

4 Assignment
Identifier **=** *expression*

The equals operator is right associative and used to assign an Identifier a constant type 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 in subsequent lines of code.

5 Data Structures

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

5.1.1 Initialization of Vectors

Vectors can be initialized using curly braces and a comma-separated list of numbers. Once initialized a vector can be assigned to an identifier:

Vector ➔ **{** *digits* (**,** *digits*)* **}**

For example the identifier x represents a vector of length 5 with the following values:

x = {1, 2,3, 4.2, 5}

An empty vector can be initialized using brackets and with an integer indicating the size of the vector. The empty vector is initialized with a value of null for each element:

Vector ➔ [digits]

For example an empty vector with a length of five:

x = [5]

5.1.2 Accessing individual 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 can be accessed using bracket index notation with an integer that represents the desired element in the vector. Integers that exceed the size of the vector -1 will throw an exception.

Vector Element ➔ *vector*[digits]

For example:

x= {1,2,3};
y = x[1];

y now points to the integer 2.

5.1.3 Binary Operators and Vectors

Binary operators can be applied to a vector and a numeric constant in which case each element of the vector has the operation applied to it resulting in a new vector. A binary operation can be applied to two vectors in which case the operation is applied to the corresponding elements of the respective vectors resulting in a new vector. Note this can only take place if each vector is the same size. Also a side effect of these operations is that some integers may be converted to floating point numbers:

Vector binary-operator (*numeric-constant* | *Vector*)

For example the following vector is multiplied by a scalar of 2:

{ 1, 2, 3} * 2
➔{ 2, 4, 6}

Similarly two vectors are added:

{1, 2, 3} + {1, 2, 3}
➔ {2, 4, 6}

5.1.4 Vector Length

Built into the language is the ability to find the size of a given vector, which will return an integer value. The syntax is as follows:

integer size ➔ *vector*.**length**

For example:

x = {1, 2, 3};
x.length ➔ 3

5.2 Matrices

Matrices are defined similarly as vectors except with multiple dimensions of numbers. In fact vectors are treated really as one-dimensional matrices.

5.2.1 Initialization of Matrices

Matrices are initialized like vectors but with an extra set of curly braces. Each set of braces represents a single row in the matrix. Each row within a matrix must have the same size:

Matrix ➔ **{ {** *digits* (**,** *digits*)* **} ,** (**{** *digits* (**,** *digits*)* **}**)* **}**

For example a 3x 3 matrix:

{{1,1,1} , {2,2,2} , {3,3,3}}

Empty matrices can be initialized using two brackets. The first bracket represents the number of rows and the second represents the number of columns:

Matrix ➔ **[** *digits* **] [** *digits* **]**

5.2.2 Accessing individual elements of a Matrix

Similarly to vectors individual elements of a matrix can be accessed using double bracket index notation, where the first bracket represents the row index and the second bracket represents the column index. An exception will occur if the indices exceed the matrix bounds:

Matrix Element ➔ *matrix* **[** *digits* **] [** *digits* **]**

For example:

x = { { 1, 2, 3 } , { 4, 5, 6 } , { 7, 8, 9 } };
x[2] [1] ➔ 8

5.2.3 Binary Operations and Matrices

Binary operators can be applied to a matrix and a numeric constant in which case each element of the matrix has the operation applied to it resulting in a new matrix. Binary operations between two matrices can be applied as long as the matrices have the same dimensions.

matrix binary-operator (*numeric-constant* | *matrix*)

For example:

{ { 1, 1, 1 } , { 2, 2, 2 } , { 3, 3, 3 } } * 2
➔ { { 2, 2, 2 } , { 4, 4, 4 } , { 6, 6, 6 } }

5.2.4 Matrix Length and Height

The length of a matrix will return an integer value representing the number of columns. The height of a matrix will return an integer value representing the number of rows. The notation is similar to that for finding the size of a vector.

integer column size ➔ *matrix*.**length**
integer row size ➔ *matrix*.**height**

5.3 Maps

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

5.3.1 Initialization of a Map

A map is initialized with a comma-separated list of key-value pairs surrounded by curly braces. The key is defined using standard string notation of double quotes. The assignment of the key to a value is done using an equals operator and a greater than operator:

Map ➔ **{** *String* **=>** *Constant* **,** (*String* **=>** *Constant*)* **}**

For example:

ages = { "danny" => 27 , "john" => 12 , "mary" => 32 } ;

5.3.2 Accessing individual elements of a map

The keys of a map are used to access individual values within the map itself. Bracket notation with the key value as a String constant is used:

Map element ➔ *map* **[** *String* **]**

For example:

ages = { "danny" => 27 , "john" => 12 , "mary" => 32 } ;
ages ["danny"]
➔ 27

Bracket notation with an integer index can also be used to access individual values of a map:

Map element ➔ *map* **[** *integer* **]**

For example:

ages = { "danny" => 27 , "john" => 12 , "mary" => 32 } ;
ages[2]
➔ 32

To access the key itself of a key-value pair the reserved keyword **key** is used taking the form:

Map element ➔ *map* **[** *integer* **].key**

ages = { "danny" => 27 , "john" => 12 , "mary" => 32 } ;
ages[2].key
➔ "mary"

5.3.3 Determine if keys exist

The map data structure has a built in method to determine if a given String is a key within the map. The method returns a Boolean value of true if the key is found or false otherwise. The syntax uses the reserved keyword **has**:

Boolean expr ➔ *map***.has(** *String* **)**

For example:

ages = { "danny" => 27 , "john" => 12 , "mary" => 32 } ;
ages.has("john")
➔ *true*

5.3.4 Find number of key-value pairs

The map data structure also has a built in method that returns the number of key/ value pairs within a map. The syntax uses the reserved keyword **length**, like the vector and matrix structures:

ages = { "danny" => 27 , "john" => 12 , "mary" => 32 } ;
ages.length
➔ 3

6 Control Structures

6.1 Statements

Statements are typically an expression with a semicolon marking the end of the statement. Unless specified statements are executed in sequence. Statements take the form of:

expression **;**

6.2 Conditional Statements

Conditional statements use a boolean expression to determine the execution of a given block of code. Conditional statements take the form of:

if (*boolean-expression* **) {** *statements* **}** (**else {** *statements* **}**) ?

The expression is evaluated and if the result is true the statements with the first set of braces are executed. If the resulting expression is false then the statements within the second set of curly braces are evaluated. The **else** block is optional.

6.3 While Statement

The while statement is a looping structure that takes the form of:

while (*boolean-expression* **) {** *statements* **}**

The Boolean expression is evaluated and if the result is true the statements within the curly braces are executed. The statements are repeatedly executed until the Boolean expression evaluates to false.

6.4 For Statement

The **for** statement is a convenience method to iterate over data structures. The for loop can be used to iterate vector, matrix and map structures taking the form:

for (*identifier* **in** *structure* **) {** *statements* **}**

Where the identifier represents the next value in the data structure.

7 Functions

Users can define any number of functions that will execute a given block of statements.

7.1 Function definition

Function definitions take the form:

function *identifier* (**(** identifier (**,** identifier)* **)**)? **{**
statements
return (*Constant*) **;**
}

The first 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 within parentheses. 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.

The last line of a function must contain the key word return followed by a constant. The constant can be a primitive constant like an integer or String, or can be a data structure. All functions must return something.

7.2 Function call

Functions can be called from the main program as part of an assignment statement or as part of an expression taking the form:

function-call ➔ *identifier* **(** (*identifier* | *constant* (**,** *identifier* | *constant*) *)? **)**

The first identifier is the name of the function being called. A function call can be passed zero or more parameters. These can be identifiers representing a constant or even another function call that returns a constant. The function will not modify a structure or assignment passed in as a parameter. If that behavior is desired assign the identifier to the result of the function.

7.3 Function example

```
function mean ( somevector ) {
        total = 0;
        for ( value in somevector ) {
                total = total + value;
        }
        return total / somevector.length;
}
```

… main program …

```
myvector = { 1 , 2, 3, 4, 5 };
average = mean( myvector );
```

average ➔ *3.0*