Language for Linear Algebra

Reference Manual

Author:	*UNI:*
Chenzhe Qian	cq2185
Guitian Lan	gl2510
Jin Liang	jl4598
Zhiyuan Guo	zg2201

October 26, 2015

Contents

1 Lexical and Syntax **3**

1.1 Identifiers . . . 3

1.2 Comments . . . 3

1.3 Keywords . . . 3

2 Data Type **3**

2.1 Scaler . . . 3

2.2 Vector . . . 4

2.3 Vector space . . . 4

2.4 Matrix . . . 4

2.5 Inner product space . . . 4

2.6 Affine space . . . 5

2.7 Array . . . 5

3 Expressions and Operators **5**

3.1 Unary operators . . . 5

3.1.1 −expression . . . 5

3.1.2 ++expression, −−expression . . . 5

3.1.3 expression´ . . . 5

3.2 Assignment operator . . . 6

3.2.1 identifier = expression . . . 6

3.3 Additive operators . . . 6

3.3.1 expression + expression . . . 6

3.3.2 expression +. expression . . . 6

3.3.3 expression − expression . . . 6

3.3.4 expression −. expression . . . 6

3.4 Multiplicative operators . . . 6

3.4.1 expression ∗ expression . . . 6

3.4.2 expression ∗. expression . . . 6

3.4.3 expression / expression . . . 6

3.4.4 expression /. expression . . . 7

3.5 Comparation operators . . . 7

3.5.1 expression <, >, <=, >=, ! =, == expression . . . 7

3.6 Linear algebra domain operators 7
3.6.1 expression @ expression 7
3.6.2 [[expression, expression]] 7
3.6.3 expression < expression, expression> 7
3.7 Precedence and Associativity 7

4 Declarations **8**
4.1 Variable Declarations 8
4.2 Function Declarations 9

5 Statements **10**
5.1 Expression statement 10
5.2 Block statements 10
5.3 Conditional statement 10
5.4 While statement 11
5.5 For statement 11
5.6 Break statement 11
5.7 Continue statement 12
5.8 Return statement 12
5.9 Null statement 13

6 Bullt-in Functions **13**

7 Sample Complete Code **14**

1 Lexical and Syntax

1.1 Identifiers

Start with a letter, followed by any letters, digits or underline _ .

1.2 Comments

Line Comments: #
Block Comments: ### ... ###

1.3 Keywords

The following are a list of reserved keywords in the language and can not be used as variable or function names.

var
vector
vecspace
matrix
inspace
affspace
affspace
function
if
else
for
while
break
continue
return

2 Data Type

In LFLA, there are four primitive data types: var, vector, matrix, vecspace, inspace and affspace.

2.1 Scaler

Primitive type **var** is a hybrid of integer and float point number.

Integer: a signed 31-bit or 32-bit unsigned integer without decimal point or exponent (see the definition of exponent in float).

float point number: it has integral part and fraction part and exponent part. The integral part can begin with an optional '+' or '-', then follows by digits. And if it is not zero, the first digit should not be '0'. The fraction part is just a decimal followed by a finite sequence of digits . The exponent part begins with 'e' or 'E', then followed by an optional '+' or '-', then sequence of digits which has non-'0' first digit. Having the fraction part, the integral part and exponent part can be missing.

If both the fraction part and exponent part are missing, then an extra decimal point should be added in the end of the integral part.

2.2 Vector

The type directly corresponds to the concept of vector in linear algebra. Formally it is a finite sequence of **var**, the length is its dimension.

2.3 Vector space

The type **vecspace** represents for the directly corresponds to the concept of vector space in linear algebra. Down to earth, it can be represented by any basis which is a maximal set of linear independent **vectors** in the vector space. In other word, it is linear spanned by a basis. Its dimension equals to the number of vectors in a basis.

2.4 Matrix

The type **matrix** directly corresponds to the concept of vector space in linear algebra. It is a two dimension array of var.

2.5 Inner product space

The type **inspace** directly corresponds to the concept of inner product space in linear algebra. It is a (v, <, >), where v is vector space type, then <, > is an inner product on the vector space span by the vectors .

An inner product is a map V*V to R satisfy following:

$$< x, y > =< y, x >$$
$$< x, x > = 0, \text{ it is zero iff } x = 0$$
$$< ax_1 + by_1, cx_2 + dy_2 > = ac < x_1, x_2 > + ad < x_1, y_2 > + bc < y_1, x_2 > + bd < y_1, y_2 >$$
$$\text{for a,b,c,d of R and } x_1, x_2, y_1, y_2 \text{in V.}$$

2.6 Affine space

The type **affspace** directly corresponds to the concept of affine space in linear algebra. It is a pair (w, V) , where w is a vector, and V is a vector space. The dimension of w should equal to the dimension of any vector in V.

2.7 Array

For any above type, there is a form in which contains multiple instances of same type, known as array. By an array of type X, we means a sequence of object of type X. Array is length fixed which means that once it is initialized, its length is immutable.

3 Expressions and Operators

LFLA contains all operators in common languages like Java and C++, except for bit manipulation operators. However, there are subtile differences between scalar-scaler, scalar-object and object-object operations and object-object operations. We will introduce these operators one by one. And in the last part of this section, we will conclude the precedence and associativity of these operators.

3.1 Unary operators

3.1.1 −expression

Return the nagetive of the expression and have the same type. The type of the expression must be **var**, **vector**, **matrix**.

3.1.2 ++expression, −−expression

The value of the expression is incremented by 1 or decremented by 1, and return the new value. Here the type of expression could only be **var** type.

3.1.3 expression´

This is the transpose operator. Return the transpose of a matrix denoted by the expression, which could only be **matrix** type.

3.2 Assignment operator

3.2.1 identifier = expression

The value of the expression is stored into the object represented by identifier. The type of identifier and expression must be the same. LFLA does not support implicit type cast.

3.3 Additive operators

3.3.1 expression + expression

This is the addition operator. For var type, it adds values of two expressions and return the new value. For vector and matrix type, it adds the elements in correspond position from two expression, and return the value of same type. The type of two expression must be the same.

3.3.2 expression +. expression

This addition operator applies to **var** with **vector**, **matrix**. It adds a var value to every elements in vector, matrix. And return the new vector or matrix object.

3.3.3 expression − expression

This is the substraction operator. Return the difference of values of two expression. It works analogously to the + operator.

3.3.4 expression −. expression

This substraction operator applies to **var** with **vector**, **matrix**. It works analogously to the +. operator.

3.4 Multiplicative operators

3.4.1 expression ∗ expression

This is the multiplication operator. For var type, it return mulplication result of two values. For matrix type, it does matrixs multiplication and return result matrix. The type of two expressions must be the same.

3.4.2 expression ∗. expression

This multiplication operator applies to **var** with **vector**, **matrix**. For two vector or matrix types, it multiply corresponding elements in two objects and return the new objects. It works analogously to the +. operator.

3.4.3 expression / expression

This is the division operator. Return the division result of two expression values or objects. It works analogously to the ∗ operator.

3.4.4 expression /. expression

This division operator applies to **var** with **vector**, **matrix**. It works analogously to the $*.$ operator.

3.5 Comparation operators

3.5.1 expression <, >, <=, >=, ! =, == expression

For these comparison operators, return 0 if it is not true and non-zero value otherwise. For **var** type, it just compare the values of expression. For **vector** type, it compares the dimension of two vectors. And it only applies to these two types.

3.6 Linear algebra domain operators

3.6.1 expression @ expression

This operator could check if a vector is in a vector space. So the left expression should be **vector** type, right expression should be **vecspace** type. If it is true, return non-zero value, otherwise return zero.

3.6.2 [[expression, expression]]

This is lie bracket operator. Two types of expressions are matrixs. Return a matrix.

3.6.3 expression < expression, expression>

This is a inner product operation. First expression is a inprod type. Second and third expression are two vector types. Return a value of var type.

3.7 Precedence and Associativity

Operators	Associativity	Precedence
[[,]],<,>	non associativity	Highest 5
´, @	left to right	4
$*$,/,$*.$,/, /.	left to right	3
+,−,+.,−.	left to right	2
<, >, <=, >=, ! =, ==	left to right	1
=	right to left	Lowest 0

4 Declarations

4.1 Variable Declarations

All variables must be declared with its data type before used. The initial value is optional. If there is one, it must be an expression resulting in the same type with variable. The grammar for variable declarator is following:

dataType identifier

Data type can be var, vector, vecspace, matrix, inspace, affspace. To declare a variable, the data type cannot be missed, and it must follows by a valid identifier. If declaring a variable with initial value, the type of value must matches the type of variable that assigned to. The following are some examples of variable declaration and initialization.

```
var v;
var v1 = 5;
var v2 = 5.1;

vector vec;
vector vec1 = [1, 2, 4.2, 5, 1.0];
vector vec2 = [v, v1, v2];

matrix mat;
matrix mat1 = [1,2.0; 3,4;];
matrix mat2 = [v, v1, v2; v, v1, v2; v, v1, v2;];
// Following is NOT allowed
// because matrix cannot interchange with vector
matrix mat3 = [vec1; vec1;];

vecspace vecsp;
vecspace vecsp1 = L(vec, vec1, vec2);
vecspace vecsp2 = L(v_1, v_2);

var vars[5];
var n = 3;
```

```
var vars[n];
vars = {1.0, 2, 3.4};
var vars[n] = {1.0, 2, 3.4};
var vars[n] = {v1, 0.2, 1, v2};

vector vecs[n];
matrix mat;
inspace insp;
inspace insp1 = inspace(vecs, mat);

affspace afsp;
vecspace vecsp3 = L(vec1, vec1, vec1);
affspace afsp1 = affspace(vec1, vecsp3);

vector vecs[n];
matrix mats[n];
inspace insps[n];
affspace afsps[n];
```

4.2 Function Declarations

A function has header and body. The function header contains function name, parameter list if any and NO need for return type. However, to declare a function it must start from keyword **function**. The name of function and names of parameters must be valid identifiers. The function body is enclosed in braces and must follow rules of statements.

The grammar for function declarator is following:
function identifier () {}
function identifier (parameter-list) {}
parameter-list: identifier, parameter-list

A simple example of a complete function definition is

```
function plus(var v1, var v2)
{
    v2 = v1 + v2;
```

```
    return v2;
}
```

5 Statements

Statements are executed in sequence.

5.1 Expression statement

The form of expression:

expression;

Most statements are expression statements. Usually expression statements are assignments or function calls.

5.2 Block statements

The form of block:

{statements}

A block encloses a series of statements by braces.

5.3 Conditional statement

Three forms of the conditional statement:

if expression {statements}

if expression {statements} else {statement;}

if expression {statements} else if expression {statements} else {statements}

In all cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case, the second substatement is executed if the expression is 0. In the third case, the second substatement is executed if the first expression is 0 and the second expression is non-zero. As usual the 'else' ambiguity is resolved by connecting an else with the last encountered elseless if.

```
var v1 = 5;
var v2 = 6;
if v1 < v2
{
    return v1;
}
else if v1 == v2
```

```
{
    return v1+v2;
}
else
{
    return v2;
}
```

5.4 While statement

The form of while statement:

while expression {statements}

The substatement is executed repeatedly as long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

```
while 1
{
    print "hello world";
}
```

5.5 For statement

The form of for statement:

for expression {statements}

The expression specifies the condition of the loop including initialization, test, and iteration step.

```
for var i = 1:5
{
    i = i + 1;
}
```

5.6 Break statement

The form of break statement:

break;

This statement causes termination of the enclosing while and for statement. It controls to pass the statement following the terminated statement.

```
for var i = 1:5
{
    if i == 2
    {
        break;
    }
}
```

5.7 Continue statement

The form of continue statement:

continue;

This statement causes control to pass to the loop-continuation portion of the enclosing while and for statement. In other words, this leads to the end of the loop.

```
for var i = 1:5
{
    if i == 2
    {
        continue;
    }
}
```

5.8 Return statement

The form of return statement:

return expression;

The value of the expression is returned to the caller of the

function. If required, the expression is converted, as if by assignment, to the type of the function in which it appears.

```
function foo()
{
    return 0;
}
```

5.9 Null statement

The form of null statement:

;

A null statement is useful to carry a label just before the } of a statement or to supply a null body to a looping statement such as while.

6 Bullt-in Functions

In LFLA language, several built-in functions are provided.

- sqrt(var x) : Returns the positive square root of a var value
  var x = 9; var result = sqrt(x); # result = 3.0

- ceil(var x) : Returns the smallest integer value that is greater than or equal to the argument
  var x = 8.8; var result = ceil(x); # result = 9

- floor(var x) : Returns the largest integer value that is less than or equal to the argument
  var x = 8.8; var result = floor(x); # result = 8

- dim(vector v) : Returns the dimension of a vector space
  vector v = [1, 2, 3]; var result = dim(v); # result = 3

- size(matrix m) : Returns the size of a matrix. Return type is an array of type var of length two
  matrix m = [1, 2; 3, 4]; result = size(m); # result = [2, 2]

- basis(vecspace vs) : Return one basis of a vector space. Return type is an array of vector
  vector v1 = [1, 0]; vector v2 = [0, 1]; vecspace vs = L (v1, v2); var[] result = basis(vs); # result = {[1, 0], [0, 1] }

- rank(matrix m) : Returns the rank of a matrix
  matrix m = [1, 2, 3; 4, 5, 6]; var result = rank(m); # result = 2

- trace(matrix m) : Returns the trace of a square matrix
  matrix m = [1, 2, 3; 4, 5, 6; 7, 8, 9]; var result = trace(m); # result = 15

- eigenValue(matrix m) : Returns the eigenvalues of a matrix. Return type is an array of var value
  matrix m = [3, 2, 4; 2, 0, 2; 4, 2, 3]; var[] result = eigenValue(m); # result = [8, -1];

- image(matrix m) : Returns the image of a matrix. Return type is vecspace
  matrix m = [1, 2; 3, 4]; vecspace result = image(m); # result = L([1, 3] , [2, 4])

- orthoBasis(inspace i) : Give the orthogonal basis of an inner product space

7 Sample Complete Code

Problem 1: Given two vectors of the same dimension, check whether they are linear independent.

```
function linearIndep(Vector[] vectors, var n)
{
    if n == 0 { return 1; }
    if n > dim(vectors[0])
        { return 0; }

    vecspace vs;
    for var i = 0:n-1
    {
        if(vectors[i]@vs)
            return 0;
        vs = vs + L(vectors[i]);
    }
    return 1;
}


main()
{
    vector v = [1,2,3];
    vector u = [2.0, 2, 4.2];
```

```
    if linearIndependent(u,v)
    {
        print "Independent";
    }else{
        print "Not independent";
    }
}
```

Problem 2: Given a inner product space ips, a basis of V, length of vector array, bases. Please orthonormalising the basis by the Gram-Schmidt method.

```
function othonomalising
    (vector bases[], inspace ips, var n){
    vector vec;
    for var i = 1:n
    {
        vec = bases[i];
        for var j = 1:i
        {
            vec = vec - <basis[i],basis[j]> *. bases[j];
        }

        bases[i] = vec /. sqrt(ips<vec,vec>);
    }
    return bases;
}

main()
{
    var n = 3;

    vector v1 = [1,2,3];
    vector v2 = [2.3, 2, n];
    vector v3 = v2;
```

```
    vector bases[n] = {v1, v2, v3};

    vector result[n] = othonomalising(bases, n);
}
```