MiniMat Language Reference Manual

Terence Lim - tl2735@columbia.edu

June 29, 2016

1 Introduction

MiniMat is a simple language to support matrix-based programming, which contains a core set of primitives that can be assembled into more complicated abstractions for matrix expressions, linear algebraic formulas and statistical algorithms. MiniMat is aimed at programmers from technical domains who work primarily with matrix expressions. It treats large data types such as a two-dimensional matrix as a primitive data type and accepts syntax so that matrices are easy to initialize, subset, combine and reshape, and all matrix operators are defined by writing functions in the MiniMat language itself.

The compiler front end components (lexer, parser, semantic analysis and intermediate code generation) are written in OCaml, OCamllex, OCamlyacc and OCaml Llvm, and translates source language into LLVM IR target code. See Chart 1.

The language reference manual is in section 2. Section 3 describes some sample programs written in the MiniMat language.

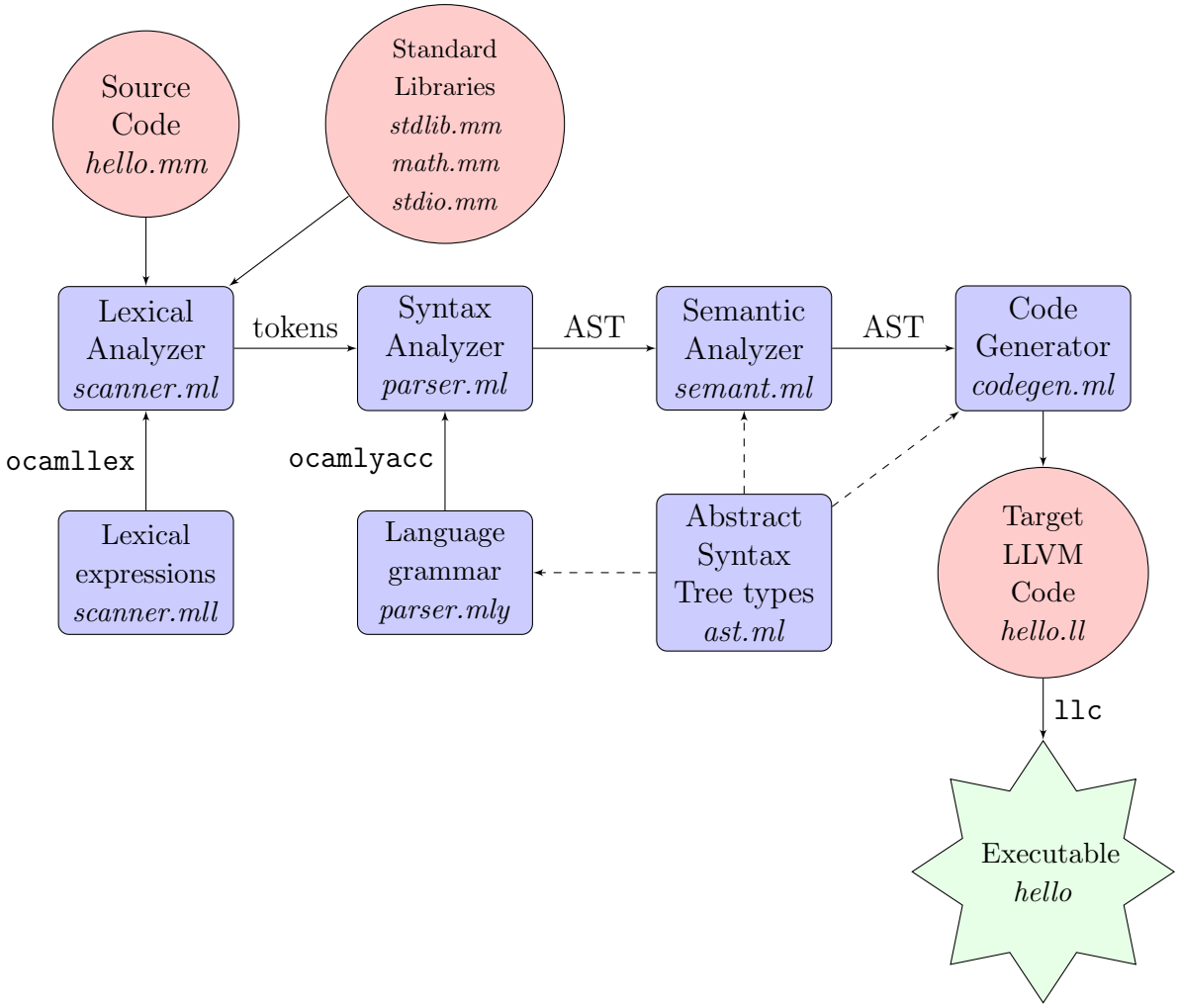
1.1 Under the hood (Architecture)

1. All matrix operators are implemented with and bound to helper functions coded in the MiniMat language itself, including arithmetic (`+ - * /`), relational (`< <= > >= == !=`), and mathematical (`inv`*erse*, `eig`*envalue*, `det`*erminant*) expressions.

2. Matrices are constructed and subselected with "matlab-like" expressions. Internally, bracketed expressions representing matrix values are parsed into lists (i.e. rows) of lists (i.e. columns) of stuff (e.g. floats or smaller matrix expressions), then the compiler's code generation component calls vertcat() and horzcat(), which are functions written in MiniMat language rather than directly in OCaml or LLVM, repeatedly on every element column-by-column and row-by-row (i.e. nested "fold left" iterations) to build up its LLVM storage representation.

3. A matrix data type is represented in the target LLVM language as an array of floats of variable size (with a `[0 x double]` struct template) allocated from the stack, prepended with a 16-byte header that identifies the actual size, #cols and #rows.

Chart 1: Block diagram of major components of translator



Hence the size of a matrix associated with an identifier can change, and is tracked automatically along with the object: the `cols`, `rows` and `length` built-in functions peek into the header stored before the data block to return the actual matrix dimensions.

4. For functions to return large data types such as matrices, the compiler takes care of temporarily copying to a block of heap memory, then back to the stack of the caller.

5. Although `floatmat` is implemented as a two-dimensional matrix of floats, it allows both row-column and "linear" index methods (see vector and matrix reference expressions in the language manual), so a `floatmat` constructed with either one row or one column can be referenced just like a one-dimensional object. Nevertheless, we retained the `intvec` data type to represent a one-dimensional vector of integers, designated for indexing purposes to access multiple elements of a matrix. Several matrix operators such as relations (which are implemented as functions written in MiniMat language) return an `intvec` of linear indices of those elements that satisfy the specified relationship, which in turn can then be used to help extract or modify those elements in the matrix.

It seemed inelegant to use `floatmat`s and have to round them to integral values to serve as indices; and more natural to separate data types of variables used for indexing versus data storage purposes.

6. Function declarations in the compiler are extended to include external function prototype declarations and global constant definitions. I felt it is very useful to be able to specify values at compile-time (e.g. for enumerations or configuration values) and to incorporate external C library functions (in particular, specialized numerical and graphical libraries like Lapack and gnuplot).

2 Language

This manual describes the MiniMat language specification.

2.1 Lexical Conventions

A program is reduced to a stream of tokens by the initial lexical analysis or scanning phase.

2.1.1 Tokens

There are five classes of tokens: identifiers, keywords, literals (of data types: integer, floating point, boolean and string), operators, and separators (e.g. parentheses, brackets, braces, semi-colons). Blanks, tabs, newlines, form feeds and comments (as described below) are ignored except to separate tokens.

2.1.2 Comments

The characters `/*` and `*/` demarcate a comment. Comments do not nest, and can be split across multiple lines.

2.1.3 Identifiers

An identifier is a sequence of letters, digits and the underscore _ character. The first character must be a letter.

2.1.4 Keywords

The following identifiers are reserved for use as keywords and implemented as built-in instructions:

```
float int void bool string if else for while return printf external constant
floatmat intvec handle sizeof length end rows cols put get new float_of_int
int_of_float
```

Additionally, matrix expressions, such as construction, assignment and selection, are implemented with helper functions written in the MiniMat language and included in the standard library file (stdlib.mm). Similarly, all matrix operators are also implemented with and bound to functions coded in MiniMat and included in the math and i/o library files (math.mm and stdio.mm).The names and descriptions of these functions are listed in the last subsection (Library functions) of this manual.

2.1.5 Literals

There are four types of literals.

- **Integer**: An integer literal, with data type `int`, consists of a sequence of digits.
- **Floating**: A floating literal, with data type `float`, consists of an optional integer part, a decimal point and a fraction part. The integer and fraction parts both consist of a sequence of digits. The integer part may be missing, and both the decimal point and fraction part are required.
- **Boolean**: A boolean literal is one of two values `true` or `false`, of data type `bool`.
- **String**: A string literal is a sequence of characters surrounded by double quotes, of data type `string`. A null byte is appended to the string C-style so that functions for printing or comparing strings can scan to find its end.

2.2 Data types

There are three *small*, three *large*, and two other data types.

2.2.1 Small types

- `int`: A signed integer has 32-bit size.
- `float`: A floating point value has 64-bit size (i.e. a *double* in C-parlance).
- `bool`: A boolean requires 1-bit size.

2.2.2 Large types

- `floatmat`: A two-dimensional matrix of floating point values, laid out in row-major order – i.e. sequential values on the same row are stored adjacent to each other. Size information, such as the number of rows and columns, is stored with the object. A one-dimensional matrix can be obtained by specifying a matrix with either one row or one column, and then using the linear indexing method to reference matrix contents: see the next subsection on matrix and vector reference expressions.
- `intvec`: A one-dimensional vector of integer values. Size information such as length is stored with the object. This data type is most helpful for providing a list of index positions to access subsets of a matrix or another vector.
- `string`: Strings are stored as null-terminated sequences of characters, up to 255 characters length (excluding terminal null).

Storage for large data types is allocated from the stack, which simplifies memory management; however, when a function exits, matrices and vectors that were newly constructed may be discarded along with the call stack frame without intervention. Hence callee functions

return large data types by temporarily copying to a block of heap memory, then back to the stack of caller: this swap is automatically implemented during compilation (code generation phase).

2.2.3 Other types

- `handle`: This data type is only utilized when calling external C library functions which may need to pass around a pointer to an external object. It is a 64-bit value representing a memory address.
- `void`: Specifying an empty set of values, it is used as the type returned by functions that generate no value.

2.2.4 Conversions

Generally, operators will not cause conversion of the value of an operand from one type to another, i.e. there is no automatic type casting. MiniMat provides two primitive built-in conversion functions `float_of_int` and `int_of_float`. When a value of a floating type is converted to integral type, the fractional part is discarded.

2.3 Expressions

The precedence of expression operators is the same as the order of the following major subsections, highest precedence first. Within each subsection, the operators have the same precedence, with left- or right-associativity as specified.

2.3.1 Primary Expressions

Primary expressions are identifiers, literals or expressions in parentheses.

An identifier is a primary expression provided it has been suitably declared as a constant; local or external function; or global, local or function argument variable. Its type is specified by its declaration.

The type of a literal depends on its form as an integer, floating, boolean or string literal.

A parenthesized expression is a primary expression whose type and value are identical to the unadorned expression.

2.3.2 Function Calls

A function call is a functional designator identifier, followed by parentheses containing a possibly empty, comma-separated list of arguments to the function. Arguments are completely evaluated before the function is entered. Recursive calls are permitted.

In preparing for the call to a function, a copy is made of *small* data type arguments: all argument-passing of `int, float, bool` and `handle` is by value. However, `floatmat, intvec` and `string` data type arguments are passed by reference: a function changing the values within its parameter objects will affect the contents of the original.

If the callee function returns a data type of `floatmat, intvec` or `string`, the compiler takes care of making a temporary copy of the returned object in a block of heap memory, then copying back to space allocated in the stack frame of the calling function.

2.3.3 Sizeof Operators

The *sizeof* functions yield the dimensions of its operand, which can be of type `floatmat` or `intvec`: `length, end, sizeof, rows, cols` return as an integer value the number of elements, last index position, number of bytes required to store, number of rows and number of columns respectively.

2.3.4 Matrix and Vector Expressions

A matrix can be defined as semi-colon-terminated rows of comma-separated floating points or smaller matrices (or their identifiers), surrounded by square brackets, of data type `floatmat`. Example: `[1.0, 2.0, 3.0; A, 6.0;]`

A vector can be defined as a comma-separated list of integers or shorter vectors (or their identifiers), surrounded by square brackets, of data type `intvec`. Example: `[1, 2, v, 3]`.

A matrix expression always ends with a semi-colon just before the closing bracket; a vector never does. A matrix requires floating point numbers, which always contain a decimal point; a vector requires integers, which do not.

A new object can be created using the `new(...)` operator: with one argument, it returns a new `intvec` with the specified length; and with two arguments, it returns a new `floatmat` with the specified number of rows and number of columns respectively. Its values are initialized to zero.

2.3.5 Matrix and Vector References

An identifier followed by two comma-separated expressions in square brackets is a postfix expression denoting a subscripted matrix reference. The two expressions together comprise the row-column index method to reference the matrix. They can each either be of `int` type (identifying a single column or row) or `intvec` type (representing several columns or rows). This row-column indexing method can be used for assigning values both to or from a subset of a matrix. Example: `A[1,2] = B[2,3]` or `A[1,[1,2,3]] = B[2,[2,4,6]]`.

An identifier followed by an expression in square brackets is a postfix expression denoting a subscripted vector reference. If the referenced operand has type `floatmat`, then the bracketed expression uses the linear index method: MiniMat treats the matrix as if its elements were strung out in a long vector by going across the rows consecutively, i.e. in row-major order. The bracketed expression can each either be of `int` type (representing a single element) or `intvec` type (identifying a subset of elements). This linear indexing method can be used for assigning values both to or from a subset of a matrix or vector. Example: `A[1] = B[2]` or `A[[1,2,3]] = B[[2,4,6]]`.

The value stored at its linear index position can also be selected with the `get(`*object,int position*`)` operator. If its first argument *object* is an `intvec`, an integer value is returned; if *object* is a `floatmat`, a floating value is returned.

A single value can be stored a linear index position with the `put(`*value,object,int position*`)` operator. The first two arguments *value,object* may either be of types `int,intvec`, or types `float,floatmat`, which respectively stores an integer value into a (integer) vector, or a floating value into a (floating) matrix.

2.3.6 Transpose operator

- **Transpose** (`'`): This is a postfix expresson that reshapes its matrix argument to transposed form, swapping its rows and columns. The operand, and hence result, are of type `floatmat`.

2.3.7 Unary operators

Unary operators are right-associative.

- **Unary Minus** (`-`): The operand must be `int, float, intvec` or `floatmat`, and the result is the negative of (all elements of) its operand.
- **Logical Negation** (`!`): The operand must have boolean type, and the result is the opposite of (i.e. *not*) the operand.

2.3.8 Power Operator

- **Power** (`^`): This operator is left-associative. When the left operand has type `float`, the right operand must have type `float` which is the power by which the left operand is raised to. The right operand `k` must have type `int` when the left operand has type `floatmat`, in which case it is matrix-multiplied by itself `k` times.

2.3.9 Multiplicative Operators

- **Multiplicative operators** (`* / %`): The multiplication, division and remainder operators are left- associative.

  The two operands must have the same type. When they are of type `float` or `int`, the binary `*` operator yields the product, while the binary `/` yields the quotient, and the `%` operator the remainder of the division of the first operand by the second; if the second operand is 0, the result is undefined.

  When the operands are of type `intvec`, the binary operators are applied pairwise by element, and the result is returned in an `intvec` of the same length.

  When the operands are of type `floatmat`, the binary `*` operator denotes matrix multiplication. The binary `/` operator yields the coefficients of a least squares regression of the left operand on the right. The binary `%` operator yields the deviations from this regression.

2.3.10 Additive Operators

- **Additive operators** (`+ -`): The addition and subtraction operators are left-associative.

  The two operands must have the same type. When they are of type `float` or `int`, the binary `+` operator yields the sum, while the binary `-` yields the difference of the two operands.

  When the operands are of type `intvec` or `floatmat`, the binary operators are applied pairwise by element, and the result is returned in an `intvec` or `floatmat` of the same size.

2.3.11 Colon Expressions

- **Colon operators with stride** (`::`): This operator takes three `int` operands and yields an `intvec` listing numbers ranging between the first to third operands, with the second operand representing the stride to skip over. The stride can be negative, in which case the first operand must be at least as large as the third operand. Example: `9:-2:1` returns an `intvector` containing odd integers between 9 and 1 inclusive in descending order. A stride value of 1 can be left out of the expression, but the two colons must be immediately adjacent. Example: `1::5` returns an `intvector` containing all integers between 1 and 5.

2.3.12 Relational Operators

- **Ordering operators** (`< > <= >=`): The less than, greater than, less or equal, and greater or equal binary operators all yield the boolean value of the specified relation, when the operands are of type `int` or `float`.

  When the operands are of type `floatmat`, the binary operators are applied pairwise element-by-element; the linear indices of pairwise elements where the specified relation is true are returned in an `intvec`.

- **Equality operators** (`== !=`): The equal and not equal binary operators all yield the boolean value of the specified relation, when the operands are of type `int`, `float` or `bool`.

  When the operands are of type `floatmat`, the binary operators are applied pairwise element-by-element; the linear indices of pairwise elements where the specified relation is true are returned in an `intvec`.

  When the operands are of type `string`, the strings' character contents are compared: strings are not equal when their respective characters in any one position index are not the same.

2.3.13 Logical Operators

- **Logical AND** (`&&`): This operator returns true if both its boolean operands are also true. It is left-associative, and guarantees left-to-right evaluation: if the first operand

evaluates to false, the value of the expression is false. Otherwise the right operand is evaluated.

- **Logical OR** (`||`): This operator returns true if either of its boolean operands are also true. It is left-associative, and guarantees left-to-right evaluation: if the first operand evaluates to true, the value of the expression is true. Otherwise the right operand is evaluated.

2.3.14 Assignment

- **Assignment operator** (`=`): The left operand must be a properly declared identifier or a selection from a matrix or vector. The value of the right expression replaces that of the object referred to by the right operand. When the right operand is an identifier with type `floatmat`, `intvec` or `string`, a new copy of its values is made and assigned to the left operand, rather than just a reference, i.e. assignment is by value.

2.4 Declarations

Declarations specify the interpretation given to each identifier. Declarations that also reserve storage are called definitions.

2.4.1 Type Specifiers

The type-specifiers are `int float bool intvec floatmat string handle void`. One type-specifier and one identifier name are given in each declaration statement.

2.4.2 Global Variables

Only global variables with *small* data type `int, float, bool` can be declared. Values of global variables can be accessed and changed by any function.

2.4.3 Global Constants

Global constant declarations have the form `constant type ID =` *literal*`;`. Only integer, floating, boolean and string literals can be assigned as values of constants. Constant values can be accessed by any function, but cannot be changed. However, their declarations are suspended when global or local variables in scope are declared with the same name: see the following subsection on Lexical Scope.

2.4.4 Functions

A function definition has the form `type ID (`*parameter-list*`) {`*body-of-statements*`}`, where the parameter list contains (possibly empty) comma separated type declarations of the function parameters.

2.4.5 External functions

External C functions can be declared and used within MiniMat programs, assuming they are loaded in the linker stage. An external function declaration has the form `type ID (`*parameter-list*`);`, where the (possibly empty) parameter list contains comma separated type declarations of the function parameters. The following list maps MiniMat data types (to *C-language external function data types*): `int` (*int32_t*), `float` (*double*), `bool` (*int8_t*), `string` (*char **), `intvec` (*int32_t **), `floatmat` (*double **), `handle` (*void **).

2.5 Statements

Statements can be of several types, and are executed in sequence.

2.5.1 Expression statements

Most statements are expression statements, such as assignments or function calls.

2.5.2 Compound statements

The compound statement, or block, can be used to execute several statements where one is expected. The bodies of a function definition or `while/for` loops are a compound statements, as is any list of statements enclosed within braces { }.

2.5.3 Selection

The `if` statement chooses a flow of control. It has two forms: `if (`*expression*`)` *statement* or `if (`*expression*`)` *statement* `else` *statement*. If the expression, which must have `bool` type when evalated, is `true` then the first substatement ie executed. In the second form, the second substatement is executed if the expression is `false`. The `else` ambiguity is resolved by connecting to the last encountered `else`-less `if`.

2.5.4 Iteration

The two forms of iteration statements specify looping.

The `while` statement has the form `while (`*expression*`)` *statement*. The substatement is executed repeatedly so long as the value of the expression remains `true`; the expression must evaluate to a `bool` type.

The `for` statement has the form `for (`*expression1 ; expression2; expression3*`)` *statement*. The first expression is evaluated once to initialize for the loop, and can have any type. The second expression must have `bool` type; it is evaluated before each iteration, and if it becomes `false`, the loop is terminated. The third expression is evaluated after each iteration, and thus specifies a re-initialization for the loop; there is no restriction on its type. The first and third expressions may be dropped, but the second expression is required.

2.6 Lexical Scope

Identifiers may specify either functions or parameters (i.e. functional arguments, local variables, global variables, and constants). The same identifier name may be used for each of these two purposes and will not interfere with one another.

The scope of a parameter of a function definition begins at the start of the block defining the function, persists through the function and ends at the end of the declarator. If an identifier is declared at the head of a function (i.e. a local variable), any declaration of the identifier outside the function as a global variable or constant is suspended until the end of the function. If a global variable is explicitly declared, then any declaration of a global constant with the same identifier name is suspended.

2.7 Library functions

This section describes the support functions coded in the MiniMat language itself that are included in several library files. These help implement all the matrix and vector expressions and operators. Only the functions listed in the first panel below are built-in: in the determination of what absolutely needs to be primitive rather than coded in the Minimat language itself, the former set generally has access to unexposed internal memory structures or calls intrinsic low-level instructions.

Built-in primitive functions:

Function	Description
void printf(string format,...)	
float float_of_int(int)	
int int_of_float(float)	
int sizeof(*object*)	
int length(*object*)	
int end(*object*)	
int rows(*object*)	
int cols(*object*)	
void put(*value*, *object*, int)	polymorphic function to assign a value into an indexed position of a matrix or vector
value get(*object*, int)	polymorphic function to select a value from an indexed position of a matrix or vector
object new(...)	polymorphic function to construct a new matrix or vector of the specified dimensions

2.7.1 Standard library: stdlib.mm

Functions for matrix expressions written in MiniMat language included in stdlib library:

Function	Description
floatmat vertcat(floatmat left, floatmat right)	helper to build matrix expression, concatenates columns to make taller matrix

floatmat horzcat(floatmat left, floatmat right)	helper to build matrix expression, concatenates rows to make wider matrix
floatmat matselect(floatmat right, intvec selectrow, intvec selectcol)	helper for matrix subset select expression
void matassign(floatmat left, intvec selectrow, intvec selectcol, floatmat right)	helper for matrix subset assignment expression
floatmat matcopy(floatmat from)	creates copy of matrix values
intvec veccopy(intvec from)	creates copy of vector values
intvec veccat(intvec left, intvec right)	helper to build vector expression
intvec vecstride(int beg, int by, int end)	helper to build colon expression. bound to :: operator.
intvec vecselect(intvec right, intvec select)	helper for vector subset select expression
void vecassign(intvec left, intvec select, intvec right)	helper for vector subset assignment expression
void linassign(floatmat left, intvec select, floatmat right)	helper for matrix subset assignment expression, linear index method
floatmat linselect(floatmat right, intvec select)	helper for matrix subset select expression, linear index method
float matget(floatmat, int)	gets floating value from a linear index position in matrix. checks that index in bounds
float matgetrc(floatmat, int, int)	gets floating value from a row-column position in matrix. checks that index in bounds
int vecget(intvec, int)	gets floating value from a linear index position in vector. checks that index in bounds
void matput(float, floatmat, int)	puts floating value into a linear index position in matrix. checks that index in bounds
void matputrc(float, floatmat, int, int)	puts floating value into a row-column position in matrix. checks that index in bounds
int vecput(intvec, int)	puts integer value into a linear index position in vector. checks that index in bounds
floatmat matnew(int row, int col)	returns a new matrix with specified dimensions. checks that row and col both positive integer values
intvec vecnew(int len)	returns a new vector with specified length. checks length is positive integer value.

2.7.2 Math library: math.mm

Functions for matrix operators and math written in MiniMat language included in math library:

Function	Description
intvec vecadd(intvec u, intvec v)	vector addition. bound to vector + operator
intvec vecsub(intvec u, intvec v)	vector subtraction. bound to vector - operator
intvec vecmul(intvec u, intvec v)	vector multiplication. bound to vector * operator
intvec vecdiv(intvec u, intvec v)	vector division. bound to vector / operator
intvec vecrem(intvec u, intvec v)	vector remainder. bound to vector % operator

intvec vecneg(intvec right)	vector unary minus. bound to vector - unary operator
floatmat matadd(floatmat u, floatmat v)	matrix addition. bound to matrix + operator
floatmat matsub(floatmat u, floatmat v)	matrix subtraction. bound to matrix - operator
floatmat matmul(floatmat u, floatmat v)	matrix multiplication. bound to matrix * operator
floatmat matdiv(floatmat y, floatmat x)	matrix division. bound to matrix / operator
floatmat matrem(floatmat y, floatmat x)	matrix remainder. bound to matrix % operator
floatmat mattransp(floatmat right)	matrix transpose. bound to matrix ' operator
floatmat matneg(floatmat right)	matrix unary minus. bound to matrix - unary operator
floatmat matpow(floatmat u, int k)	matrix subtraction. bound to matrix - operator
floatmat inv(floatmat v)	matrix inverse
intvec matlt(floatmat u, floatmat v)	matrix less than. bound to matrix < operator
intvec matle(floatmat u, floatmat v)	matrix less or equal. bound to matrix <= operator
intvec matgt(floatmat u, floatmat v)	matrix greater than. bound to matrix > operator
intvec matge(floatmat u, floatmat v)	matrix greater or equal. bound to matrix >= operator
intvec mateq(floatmat u, floatmat v)	matrix equal. bound to matrix == operator
intvec matne(floatmat u, floatmat v)	matrix not equal. bound to matrix != operator
float det(...)	determinant
float cond(...)	condition number
int rank(...)	rank
floatmat eig(...)	eigenvalue decomposition
floatmat svd(...)	singular value decomposition
floatmat lu(...)	LU factorization
floatmat qr(...)	QR factorization
floatmat chol(...)	Cholesky factorization
floatmat rref(...)	reduced row echolon form
floatmat adjoint(...)	adjoint

2.7.3 I/O library: stdio.mm

Functions for input/output and type conversions written in MiniMat language included in stdio library.

Function	Description
bool stringeq(string a, string b)	string equals. bound to string == operator
bool stringne(string a, string b)	string not equals. bound to string != operator
float float_of_string(string s)	converts string to float
int int_of_string(string s)	converts string to int
intvec vec_of_int(int i)	converts int to intvec
int int_of_vec(intvec v)	converts vecint[0] to int
floatmat mat_of_float(float i)	converts float to floatmat
float float_of_mat(floatmat v)	converts floatmat[0] to float
floatmat mat_of_vec(intvec v)	converts vec to mat
intvec vec_of_mat(floatmat v)	converts mat to vec

void printbool(bool b)	prints a boolean literal
void printfloat(float f)	prints a floating literal
void printvec(intvec v)	prints an intvec
void printmat(floatmat v)	prints a floatmat
string next()	returns next string, up to 255 characters long, from stdin; or empty string `""` at end of input. Ignores white space.
void print(int)	
void printb(bool)	

3 Sample Programs

3.1 Expressions

This sample program illustrates how to write expressions in the MiniMat language to:

1. Initialize a new matrix
2. Augment matrices
3. Select a subset from a matrix
4. Assign to a subset of a matrix

Listing 1: MiniMat program to manipulate matrix expressions

```
int main() {
  floatmat Z;
  floatmat Y;
  floatmat X;
  intvec I;

  /* Create a size=3 identity matrix */
  X = matnew(3,3);
  I =  0:cols(X)+1:end(X);       /* linear indexes of its diagonal */
  X[I] = [1.0;];                 /* assign ones to diagonal */
  printf("Identity matrix:\n");
  printmat(X);

  /* Create a 3X3 matrix of two */
  Y = matnew(3,3) + [2.0;];
  printf("Matrix of 2's:\n");
  printmat(Y);

  /* Populate matrix with increasing values counting from 1.0 by 2.0*/
  Z = matnew(3,3);
  Z[0::end(Z)] = mat_of_vec(1:2:2*length(Z));
  printf("Matrix with increasing numbers by 2.0:\n");
  printmat(Z);

  /* Stitch together four corners of new matrix */
  printf("Large stacked matrix:\n");
  printmat([X, Y; Y, Z;]);
}
```

Output Log

```
Identity matrix:
  1.00 0.00 0.00
  0.00 1.00 0.00
```

```
  0.00 0.00 1.00
Matrix of 2's:
  2.00 2.00 2.00
  2.00 2.00 2.00
  2.00 2.00 2.00
Matrix with increasing numbers by 2.0:
  1.00 3.00 5.00
  7.00 9.00 11.00
 13.00 15.00 17.00
Large stacked matrix:
  1.00 0.00 0.00 2.00 2.00 2.00
  0.00 1.00 0.00 2.00 2.00 2.00
  0.00 0.00 1.00 2.00 2.00 2.00
  2.00 2.00 2.00 1.00 3.00 5.00
  2.00 2.00 2.00 7.00 9.00 11.00
  2.00 2.00 2.00 13.00 15.00 17.00
```

3.2 Functions

This sample program illustrates more examples of building up programming abstractions with the MiniMat language:

1. Define a new mathematical operator function (note that all standard matrix operators are defined in this same way using functions written in MiniMat language itself) – the division operator `/` for matrices.

2. Define a new linear algebraic function – to compute the inverse `inv` of a matrix argument.

3. Declare and use external functions – from the Lapack C API library.

4. More MiniMat matrix expressions: we set up the values of operands for a statistical formula, to compute a linear regression with constant intercept.

5. Write a (one-line) statistical program in MiniMat – to identify outlier observations when fitting a linear regression model.

There is no standard mathematical definition of the matrix division operator `/`, so we shall provide our own by coding it up a function in the MiniMat language which shall be incorporated into its math library file `math.mm`. We define the operator to return the least square regression coefficients $Y/X = \beta = (X'X)^{-1}X'Y$, so that with scalar operands for example, the quotient β exactly satisfies $Y = \beta X$, though only approximately satisfies (but with minimum squared error) $Y \simeq \beta X$ with matrix operands. The regression formula requires computing a matrix inverse, so we write such a MiniMat function, that uses specialized routines from the external Lapack numerical library.

The final program to compute outlier observations from a linear regression model can be specified in one line that aesthetically resembles an intuitive matrix equation.

Listing 2: MiniMat program to define and use functions

```
1 /* Sample program to fit linear regression model and display outliers */
2
```

```
/* Use LAPACK external C API library */
constant int LAPACK_ROW_MAJOR = 101;
external int LAPACKE_dgetrf(int M,int m,int n,floatmat A,int lda,intvec pivot);
external int LAPACKE_dgetri(int M,int m,floatmat A,int lda,intvec pivot);

/* Define matrix inverse function, with the help of LAPACK */
floatmat inv(floatmat v) {
  floatmat A;
  int info;
  intvec pivot;
  A = matcopy(v);
  pivot = vecnew(rows(A));
  info = LAPACKE_dgetrf(LAPACK_ROW_MAJOR, rows(A), cols(A), A, rows(A), pivot);
  info = LAPACKE_dgetri(LAPACK_ROW_MAJOR, rows(A), A, rows(A), pivot);
  return A;
}

/* Define matrix division function, matdiv is bound to matrix / operator */
floatmat matdiv(floatmat y, floatmat x) {
  return inv(x' * x) * (x' * y);
}

/* Main program: generate some values and find outliers */
int main() {
  floatmat Y;
  floatmat X;
  floatmat outliers;

  /* arbitrarily create a column of Y values and two colums of X values */
  Y = [2.0; 0.5; 1.5; 5.0; 7.0; 7.0;];
  X = [1.0,2.0; 2.0,2.0; 3.0,3.0; 4.0,3.0; 5.0,5.0; 6.0,6.0;];

  /* insert a column of ones at front of X, for regression intercept */
  X = [matnew(rows(X), 1) + [1.0;], X;];

  /* find and display large negative outliers from regression model fit */
  outliers = Y[Y - X*(Y/X) < [-1.0;]];

  printmat(outliers);
}
```

References

- Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 2006. Second Edition.
- Brian W. Kernighan, and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 1998. Second Edition.
- Stephen A. Edwards. *The MicroC Compiler.* Columbia University COMS4115 lecture slides.