Mattricks

Language Reference Manual

Manager	Yuxin Yang
System Architect	Weisheng Wang, Yuanhan Tian
Language Guru	Jack Wang, Danny Hou
Tester	Hang Yuan, Yuhao Dong

Table of Contents

1. Introduction

Mattricks is an imperative programming language focused on mathematical calculations similar to Matlab, with a particular focus on matrix manipulation. The motivation behind our language is that matrix calculation is widely used in linear algebra and machine learning models, and there is no existing language that provides the perfect solution for such calculation. Existing languages either need the support of additional libraries or their complicated grammar makes it less friendly for new users to perform matrix calculation. Therefore, we want to design a language that provides users with the ability to perform matrix calculations with peace of mind. In Mattricks, we will include mathematical symbols and syntax, so users do not need to install any math packages. Moreover, the matrix syntaxes are designed to be easy to understand for users, so users without much prior knowledge of computer science will also be able to write their own calculations using our language. It also provides more readability so that users can easily understand the programmers' intentions.

Static Scoping could improve the readabilities and help users to avoid confusion about where each variable (number) is accessed from. It also requires less run-time overhead. Strict evaluation specifies the order of executing different function arguments to avoid side-effects. A strongly and statically typed system sets everything clear to users on what type of elements they are using and allows less confusion. With such paradigms and features incorporated into itself, Mattricks gives the users a way to perform small to large scale matrix calculations with more efficiency and flexibility.

2. Lexical Conventions

There are 5 types of tokens: identifiers, keywords, literals, expression operators and separators.

2a. Identifiers

Identifiers are sequences of alphanumeric characters plus the underscore(a-z, A-Z, 0-9, _). The leading character of the identifier must be either a letter or an underscore. Keywords are special identifiers and are reserved for special usage, so normal identifiers and keywords are mutually exclusive.

```
let letter = ['a'-'z' 'A'-'Z'];;
let digit = ['0'-'9'];;
let underscore = ['_'];;
let id = (underscore | letter) (underscore | letter | digit )*;;
```

2b. Keywords

Keywords are special identifiers and are reserved for special usage.

```
int
float
bool
function
gives
return
console
consolef
mat
true
```

```
True
false
False
and
or
while
if
else
```

2c. Literals

Since Mattricks targets matrix computation, its two common classes of literals are integers and floats.

- Integers are sequences of numbers starting with a nonzero number.

```
let nonzero = ['1'-'9'];;
let zero    = ['0'];;
let int     = (nonzero) (nonzero | zero)*;;
```

- floats are sequences of numbers and have exactly a dot('.'). Floats' integer part should be a valid number and floats may terminate by dot.

```
let nonzero      = ['1'-'9'];;
let zero         = ['0'];;
let dot          = ['.'];;
let valid_number = (nonzero (nonzero | zero)*) | zero;;
let float        = (valid_number) (dot) (nonzero | zero)*;;
```

- Boolean is a built-in expression of Mattricks. The value of boolean is either true or false. To simplify the expression, we allow the true or false written in both uppercase and lowercase start. Like C/C++, the boolean expression is interchangeable with numerics zero and non-zero to simplify the calculations such as the activation function of the neural networks.

```
let boolean = ['true' 'false' 'True' 'False'];;
```

2d. Operators

Operators are tokens that have been reserved for computation. Mattricks provide 3 types of operators: arithmetic, assignment and comparison. More details can be found at the Operators section.

- Arithmetic operators
- ‘+’, when applied to the numerics, the program will perform numeric addition. When the operator is applied on matrices, the program will perform matrix addition.
- To be noted, when performing matrix operations, the matrix must be in the correct dimensions, which means that they must be feasible in terms of mathematics. Otherwise, the program may not be able to function correctly.
- Stream operators will be used to print out the content of variables. The program will print out the content of that variable in the console.
- Other operations are similar to the specs of addition operation.

```
let arithmetic_op = ['+' '-' '*' '/' '%'];;
```

- assignment operators

```
let assignment_op = ['='];;
```

- Comparison operators

```
let equivalence _op = ['==' '!=' '>' '>=' '<' '<='];;
```

- Stream operators

```
let stream_op = ['<<'];;
```

2e. Separators

The separators include blanks, newlines, tabs, and comments (if the characters "/**/" are not in a string, then it means to start a comment and will be considered as a separator).

```
let sep = [' ' '\t' '\r' '\n'];
```

3. Types

3a. Primitive Data Types

int

The int type is a built-in primitive data type.

It stores the integer value in a 32-bit int.

Explicit declaration

```
a = int 1;
```

float

Float is a built-in primitive data type.

It stores the float-precision floating point value in a 32-bits float.

Explicit declaration

```
a = float 0.4;
```

bool

Boolean is a built-in primitive data type. It stores the binary value, which can be either true or false. To be noted, the int value can be casted to boolean to facilitate the arithmetic operations.

Integer casting

```
a = bool 1; /* a is true */
b = bool 0; /* a is false */
c = bool 3; /* c is also true */
```

Explicit declaration

```
a = bool true;
a = bool false;
a = bool True;
```

3b. Non-Primitive Data Types

Mat

Mat be a built-in, non-primitive data type of Mattricks. It is a fixed-sized, fixed dimensioned data structure representing matrices

The syntax is <var> <assignment_op> <type> (<dimension>) <content>.

Our language support the declaration of matrices with more than 2 dimensions.

However, we did not have the time to support the subscript access of matrices with dimensions greater than 2.

Example

```
Vec       = mat int [2];
FloatVec = mat float [1];
BoolVec   = mat bool [3];
NestedVec = mat mat int [3] [2];
```

Declaration Grammar

```
(* declaration *)
| ID ASSIGN one_d_array_rule SEMI { DeclareOneDArray ($1, $3) }
```

```
(* access *)
| ID LBRAC expr_rule RBRAC ASSIGN expr_rule { OneDArrayAssign ($1,
$3, $6) }

| ID LBRAC expr_rule RBRAC { ArrayAccess($1, $3) }

| ID LBRAC expr_rule RBRAC LBRAC expr_rule RBRAC { AnyArrayAccess($1,
$3, $6) }

| ID LBRAC expr_rule RBRAC LBRAC expr_rule RBRAC ASSIGN expr_rule {
TwoDArrayAssign ($1, $3, $6, $9) }
```

Array subscript access

```
Vec = mat mat int [1][2];
a  = Vec[1][2]; /* valid */
a  = Vec[1, 2]; /* invalid */
```

3c. Function

Our program support declaring functions with customized return types.

```
function add(int a, int b) gives int {
    return a + b;
}
```

Function declaration grammar

```
fdecl_rule:
  FUNCTION ID LPAREN formals_opt RPAREN GIVES typ_rule LBRACE
vdecl_list_rule stmt_list_rule RBRACE
  {
    {
      rtyp=$7;
      fname=$2;
      formals=$4;
      locals=$9;
      body=$10
    }
  }
```

3d. Type Casting

The primitive type variable can be statically cast to another type and type casting must be explicitly noted. For instance, we can console both boolean and integer type values.

Syntax

```
A = int 1;
B = bool true;
console << A;
console << B;
```

4. Operators

4a. Arithmetic

The binary operation supports 6 different kinds of operation.

```
'+', '-', '*', '/', '%'
```

The first five operators' precedence follow the same precedence as they are in C.

Example

```
A  = 2;
a = int 1;
a = a + 1 - 2 * 3 / 4 % 5;
```

Grammar

```
expr_rule:
  | expr_rule TIMES expr_rule     { Binop ($1, Times, $3 )}
  | expr_rule DIVIDE expr_rule    { Binop ($1, Divide, $3 )}
  | expr_rule PLUS expr_rule      { Binop ($1, Add, $3)   }
  | expr_rule MINUS expr_rule     { Binop ($1, Sub, $3)   }
  | expr_rule MODULUS expr_rule   { Binop ($1, Modulus, $3) }
```

4b. Assignment

There are two types of assignment operators in the language, declaration assignment and assignment.

Declaration assignment operator does not specify the type of variables being assigned.

Assignment operator does specify the type of variables being assigned. The type will need to be specified at the right hand side of assignment.

Syntax

```
A = 1;
B = 2.0;
C = true;
D = int 1;
E = float 2.0;
F = bool true;
G = mat int [1];
H = mat float [1];
I = mat bool [1];
J = mat mat int [1][1];
```

Grammar

```
(*Declaration and Assignment*)
| ID ASSIGN typ_rule expr_rule SEMI          { BindAssign ($3, $1, $4) }
| ID ASSIGN INTMAT LPAREN LITERAL COMMA LITERAL RPAREN SEMI { DeclareMat($1, $5,
$7) }
```

4c. Equivalence

==, !=, >, <, >= and <= are used to compare the value.

They are right associative.

The return of equivalence expressions is boolean or int 0 or int 1.

Syntax

```
C = 1 < 2;  /* false */
```

```
D = 3 == 3; /* true  */
F = 6 >= 3; /* true  */
```

Grammar

```
expr:
   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)     }
```

4d. Logical

The AND and OR operation is supported in this language. They are used in loops and if-else statements. AND has higher precedence than OR.

Syntax

```
if (1 == 1 and 2 == 3 or 1 == 1) {
 /* The condition evaluate to True,
  so this block Will be executed*/
}
```

Grammar

```
expr:
  expr AND expr { Binop($1, And, $3) }
| expr OR expr { Binop($1, Or, $3) }
```

4f. Stream

The stream operator is an operator used to print the data to the console. It is paired with the “console” keyword. The right hand side of the stream operator is the data which will be sent to the console.

Syntax

```
x = int 5;
y = float 1.0;
console << x;
console << y;
/* 5
   1.0
*/
```

Grammar

```
| CONSOLE PRINTF expr_rule      { Printf $3             }
| CONSOLEF PRINTF expr_rule     { FPrintf $3            }
```

5. Statements and Expressions

5a. Literal Expressions

Each data type as mentioned previously could stand alone as an expression. Both primitive and non-primitive data types could be alone as an expression in Mattricks, and also could be participated in comparison / assignment with other expressions.

Syntax

```
12.56; /* float*/
100;   /* integer*/
True;  /* boolean*/
false; /* boolean*/

12.56 <= 23.45; /* literals participate in comparison */
b = int 12;     /* literals participate in assignment*/
```

Example above shows that int, float, boolean could stand alone as an expression, and could join other expressions and/or assignments to form a bigger expression.

5b. Declarations

In Mattricks, variables could be declared through either an implicit or an explicit declaration.

The syntax for the explicit declaration is that we start with a variable name, then followed by an assignment_op (=), then followed by the data type it will be assigned to, and lastly the value to be assigned to the variable.

Syntax

```
var_name = int 12;
_value   = float 12.33;
vec = mat int [1];
vec2d = mat mat int [1][1];
```

In the case that an expression contains multiple assignment_ops (e.g. multiple = in the expression), the rightmost assignment_op will be evaluated first and then continue to evaluate the rest in the order from the right to the left.

Syntax

```
a = b = int 12;
```

5c. Array and Matrix expressions

In Mattricks, the syntax of a matrix expression declaration starts with the similar syntax for the declaration as the variable declaration – starts with the variable name, followed by assignment_op, and then the data type it assigns to. Moreover, matrix can only store the elements/values with the same data type. For the syntax, after the variable name, declaration_assignment_op and the type, there is a [x] indicating the size of the matrix – [x] means the matrix will have x rows and y columns. After indicating the rows and columns, we then can list the values in the [ ] brackets.

Syntax

```
vec = mat int [1];
vec2d = mat mat int [1][1];
/*
equivalent to cpp
    vector<int>v(1);
    vector<vector<int>>v2d(1, vector<int>(1));
*/
```

In Mattricks, array is a special 1 x n type of matrix following exactly the same rules as described above.

The initial value of matrices is undetermined.

User should assign the value to the matrix after the declaration.

For example:

```
int i;

function init_val_1d_int_10(mat int [10] m, int val) gives mat int [10] {
   i = 0;
   while (i < 10) {
       m[i] = val;
       i = i + 1;
   }
   return m;
}

function print_1d_int_10(mat int [10] m) gives int {
   i = 0;
   while (i < 10) {
       console << m[i];
       i = i + 1;
   }
   return 0;
}

function addval_1d_int_10(mat int [10] m, int val) gives mat int [10] {
   i = 0;
```

```
    while (i < 10) {
        m[i] = m[i] + val;
        i = i + 1;
    }
    return m;
}

function main(int x) gives int{

    mat_1d_int_10 = mat int [10];

    /* init the 1D matrix with 0 */
    mat_1d_int_10 = init_val_1d_int_10(mat_1d_int_10, 0);
    print_1d_int_10(mat_1d_int_10);

    /* init the 1D matrix with 123 */
    mat_1d_int_10 = init_val_1d_int_10(mat_1d_int_10, 123);
    print_1d_int_10(mat_1d_int_10);

    /* minus 123 to all the values in the 1D matrix */
    mat_1d_int_10 = addval_1d_int_10(mat_1d_int_10, -123);
    print_1d_int_10(mat_1d_int_10)
}
```

5d. Printing Matrix

The keyword for printing is console followed by <<, and then followed by the variable name / values that need to be printed. In addition, the matrix could be printed as well, and the print format for the matrix is uniquely defined.

Syntax

```
mat = mat mat int [3][3]
mat[1][1] = 123;
console << mat[1][1];
/* result: 123 */
```

5e. Function for matrix

In Mattricks, to define a function, we need to include the keyword function and the variable name for the function. After that, we need the parenthesis. Inside the parenthesis, we need to list the arguments that need to be passed into the function. For the arguments, they need to follow the same rule as the variable declaration that we discussed previously. After we define the arguments, there is another keyword `gives` that defines the return type of the values this function returns. After the `gives` es keyword, we will show the return type as mentioned above. The main body of the function has to be inside the curly brackets. The keyword `return` is used to identify the value/variable the function returns.

Syntax

```
function function_name (argument1, argument2, ...) gives return_type
{
   /* function body */
   return return_value;
}
```

The function can also take multiple arguments and the returned value can be assigned to other variables as well

Syntax

```
function f(mat mat int [3][3] p, mat mat int [3][1] q) gives mat mat int [3][1] {
  return p * q;
}
/* variables that are going to pass into the function */
mat1 = mat mat int [3][3]
mat2 = mat mat int [3][1]
mat3 = mat mat int [3][2]
/* passing the variables into the function and assign mat3 to the
   returned value from the function defined above */
mat3 = f(mat1, mat2);
```