Sophie Reese-Wirpsa (smr2225)
Diego Prado (dtp2118)
Cindy Espinosa (cje2127)
Desu Imudia (aii2003)
Emily Ringel (edr2124)

Language Reference Manual

MatrixMania

PLT Spring 2021

1. Introduction

The goal of our language is to ease the manipulation of matrices. A few features of our

mathematical language are built-in matrix addition, subtraction, and multiplication as well as array programming.[1] Array programming would allow the user to apply operations of a primitive against an entire matrix, an entire row of a matrix, or an entire column of a matrix. Our language will rely primarily on a matrix data structure, and to complement this we will replace strings and arrays with 1-dimensional matrices. We plan to use python-like syntax and static typing and will also be enabling type-hinting in order to better indicate the type values within our code. Our language will be derived from Java.

Our language will be used to write programs to manipulate and solve matrices. These programs could include algorithms to invert an n-by-n matrix or to put a matrix into reduced row echelon form by Gauss-Jordan Elimination. These types of algorithms would benefit from many of the features in our language, such as the built in ability to multiply a matrix or part of a matrix by a primitive type value (by array programming) or syntax to make indexing more readable. Array programming, for example, would simplify the steps of Gauss-Jordan elimination where it is necessary to multiply a row by a scalar by avoiding the use of a loop. Our language will also replicate the basic operations and types from Java, which will make it possible to run most basic algorithms, including GCD and Hello World.

2. Lexical Conventions

   Included in MM are six types of tokens: identifiers, keywords, constants, strings, expression operators, and other separators.

   a. Comments

      Comments begin with /* and end with */ such that any text between them will be ignored.

   b. Identifiers (Names)

An identifier is a sequence of letters and digits that must begin with an alphabetic character. Upper and lower case letters are distinct.

c. Keywords

The following identifiers are reserved for use as keywords:

```
return
break
continue
if
else
for
while
matrix
import
define
def
none
```

d. Constants

i. Integer constants

An integer constant is a sequence of digits.

ii. Float

A floating constant consists of an integer part, a decimal point, a fraction part, an e, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the

e and the exponent (not both) may be missing.

3. Syntax Notation

   The syntax notation in this manual includes…

   a. syntactic categories indicated by *italic type*

   b. literal words and characters in `source code mono`

   c. alternatives listed on separate lines

   d. optional terminal or non-terminal symbol is indicated by the subscript ‘‘opt,’’

      Example:

      $\{expression_{opt}\}$

4. Objects and lvalues

   a. An object is a mutable region of storage .

   b. An lvalue refers to an object that can be assigned to; it is an object that persists beyond a single expression. The left operand must be an lvalue expression.

5. Conversions

   a. Float; integer

      All `ints` may be converted without loss of significance to `float`. Conversion of `float` to `int` takes place with truncation towards 0.

6. Expressions

   The precedence of expression operators is the same as the order of the major subsections of this section (highest precedence first). Operators within each subsection have the same precedence.

   a. PRIMARY EXPRESSIONS

      Involve and function calls group left to right

i. IDENTIFIER

Identifiers are primary expressions. Identifiers are symbols which name the language entities (listed in keywords of the lexical conventions section 2c). Its type is specified by its declaration.

ii. CONSTANT

An integer or floating constant is a primary expression.

iii. { EXPRESSION }

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

b. UNARY OPERATORS

Expressions with unary operators group right-to-left.

i. ! expression

The result of the logical negation operator ! is 1 if the value of the expression is 0, 0 if the value of the expression is non-zero. The type of the result is `int`. This operator is applicable only to `ints`.

c. MULTIPLICATIVE OPERATORS

The multiplicative operators *, /, and % group left-to-right

i. *expression* * *expression*

The binary * operator indicates multiplication. If both operands are `int`, the result is `int`; if one is `int` and one `float`, the result is `float`; if both are `float`, the result is `float`.

ii. *expression* / *expression*

The binary / operator indicates division. The same type considerations as for * (multiplication) apply.

iii. *expression* % *expression*

The binary % operator yields the remainder from the division of the first expression by the second. Both operands must be `int`, and the result is `int`. In the current implementation, the remainder has the same sign as the dividend.

d. ADDITIVE OPERATORS

The additive operators + and − group left-to-right.

i. *expression* + *expression*

The result is the sum of the expressions. If both operands are `int`, the result is `int`. If both are `float`, the result is `float`. If one is `int` and one is `float`, the result is `float`.

ii. *expression* − *expression*

The result is the difference of the operands. If both operands are `int` or `float`, the same type considerations as for + (addition) apply.

e. RELATIONAL OPERATORS

The relational operators group left-to-right.

i. *expression* < *expression*

ii. *expression* > *expression*

iii. *expression* <= *expression*

iv. *expression* >= *expression*

The operators < (less than), > (greater than), <= (less than or equal to) and

>= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. Operand conversion is exactly the same as for the + operator.

f. EQUALITY OPERATOR

   i. *expression* == *expression*

   ii. *expression* != *expression*

   The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus ‘‘a < b == c < d’’ is 1 whenever a<b and c<d have the same truth-value)

g. LOGICAL OPERATORS

   i. *expression* && *expression*

   The && operator returns 1 if both its operands are non-zero, 0 otherwise. Guarantees left-to-right evaluation. The operands need not have the same type, but each must have one of the fundamental types (`int` or `float`).

   ii. *expression* || *expression*

   The || operator returns 1 if either of its operands is non-zero, and 0 otherwise. Guarantees left-to-right evaluation. The operands need not have the same type, but each must have one of the fundamental types (`int` or `float`)

h. ASSIGNMENT OPERATORS

Group right-to-left. All require an lvalue as their left operand. The value is the value stored in the left operand after the assignment has taken place.

   i. *lvalue* = *expression*

   The value of the expression replaces that of the object referred to by the

lvalue. The operands need not have the same type, but both must be `int` or `float`. If operands are different types (`float` and `int`) the assignment takes place as expected, preceded by conversion of the expression on the right. When both operands are `int` or `float`, no conversion takes place; the value of the expression is simply stored into the object referred to by the lvalue.

7. Statements

   a. Expression Statements

      Most statements are expression statements and take the form

      `expression ;`

      They can be assignments or function calls.

   b. Conditional Statements

      Options for the conditional statement are

      `if` (*expression*) {*statement}*

      `if` (*expression*) {*statement}* `elif` (*expression*) ... {*statement}* `else` {*statement}*

      The expression is evaluated and if it is non-zero, then the first substatement is executed. For the second case, if the first expression is zero then the second substatement is checked, if it is non-zero, then the substatement is executed. If the second statement is zero, then the substatement connected to the `else` block is executed. One can have as many `elif` blocks between the `if` block and the `else` block.

   c. While statement

The `while` statement takes the form

`while` (*expression*) {*statement}*

The statement is executed while the expression evaluates to a non-zero value. The test of the expression occurs before the execution of the statement.

d. For statement

The `for` statement takes the form

`for` (*expression-1*; *expression-2*; *expression-3*) {code block: *statement*}

Expression 1 is executed once before the execution of the statement in the code block. This expression specifies the initialization for the loop.

Expression 2 defines the condition for executing the statement in the code block. The test of the condition occurs before each iteration and the loop is excited when the expression evaluates to 0.

Statement 3 is executed every time after the statement in the code block has been executed. It typically specifies an incrementation of the value initialized in expression 1.

e. Break statement

The statement

`break;`

ends the smallest enclosing loop (while or for) .

f. Continue statement

The statement

`continue;`

causes control to pass to the loop-continuation of the smallest enclosing loop

(while or for). The control passes to the end of the loop.

g. Return statement

A function uses the return statement to return to its caller. It can take one of the following forms

`return ;`

`return` (*expression*) ;

8. Scope Rules

Variables can be defined inside and outside of functions. For variables that are defined inside of a function, they are bound within the brackets of the function they are defined in. For variables defined outside of functions, they are "global" variables and can be used throughout the file.

9. Compiler Control Lines

When a line of MM begins with the keyword "import", it is interpreted not by the compiler itself but by a preprocessor. The preprocessor replaces instances of given identifiers with arbitrary token-strings and inserting named files into the source program.

a. Token replacement

A compiler-control line of the form

`define` *identifier token-string* ;

b. File inclusion

When wanting to include several files to create a larger program, a compiler control line of the form

`import` "filename" ;

This results in the replacement of that line by the entire contents of the file

filename.

10. Implicit Typing

    MM will use dynamic typing–variables will not be explicitly declared with their types. The type of the variable will be deduced at compile time by the compiler using the variable used to initialize the variable.

11. Matrices

    Matrices will be constructed by the user giving values for the rows and columns. A semicolon will be used to distinguish between rows. Commas will be used to distinguish between values in each row (differing columns).

    For example:

```
m = [8, 1; 17, 4];
```

    a. Intrinsic Operations:

        i. Size of matrix

        Return value is a matrix in the form [*rows*, *columns*]

```
m = [1, 2; 3, 4];
sizeOfMatrixM = size m;
```

        ii. Indexing into rows, columns

```
m[2,2]
```

    b. Operations:

        i. Add, Subtract

```
m = [1, 2; 3, 4];
m2 = [1, 2; 3, 4];
newMat = m + m2;
m2 = newMat - m;
```

```
/*adding and subtracting to specific rows or
   columns*/
newMat2 = 2 + m[2 ; null];
newMat3 = 3 + m[null ; 1];
```

ii. Multiply

```
m = [1, 2; 3, 4];
m2 = [2, 4; 6, 8];
m2 = m*2;
matrixNew = m * m2;
/*multiplying specific rows or columns*/
newMat = 2* m[2 ; null];
```

12. Constant Expressions

    A constant expression is an expression that contains only constants. A constant expression can be evaluated during compilation rather than at run time, and can be used in any place that a constant can occur.

    Some places may require expressions that evaluate to a constant (after `case`, array bounds, initializers) and may be connected by binary operators when involving integer constants.

13. Examples

    Each code sample details some capabilities of our language and also the use of eventual standard library functions of our `matrix` type such as *getMatrixDeterminant()* and *getMatrixInverse()*.

Inverting a Matrix:

The sample code below includes a main executable function that creates a matrix m and prints out the resulting matrix after inverting the original when given a 2x2 integer matrix.

```
def getMatrixDeterminant(m){
   return (m[1,1] * m[2,2]) - (m[1,2] * m[2,1]);
}

def getMatrixInverse(m){
   determinant = getMatrixDeterminant(m);
   if (determinant == 0){
      return [-1];
   }
   return 1/determinant * [m[2,2], -1*m[1,2]
                           -1*m[2,1],  m[1,1]];
}

def main(){
    m = [1, 2; 3, 4];
    print(getMatrixInverse(m));
}
```

Transpose a Matrix:

The sample code below creates the transpose of a matrix by changing the rows to columns and columns to rows. It takes in a populated matrix n and stores the results in an empty matrix m.

```
def transpose(m, n){
   for(i = 0; i < size n ; i + 1){
      for(j = 0; j < size n; j + 1){
         m[i,j] = n[j,i];
      }
   }
}
```

Greatest Common Divisor:

The sample code below uses a while loop to find the GCD of two integer inputs a and b. We are subtracting the smaller number from the larger number until they both become the same. At the end of the loop the value of the numbers would be the same and that value would be the GCD of these numbers.

```
def gcd(a, b) {
```

```
   num1=a;
   num2=b;

   while(num1!=num2){
      if(num1>num2){
         num1 = num1 - num2;
      }
      else{
         num2 = num2 - num1;
      }
   return num2;
}
```