Matrix Entertainment Langauage (MatrEL)

Language Reference Manual

Rochelle Palting
<rcp2122>
Columbia University

1 Introduction

MatrEL is a computer language designed for board game creation. Examples of games that can be created are Tic-Tac-Toe, Minesweeper, and Battleship. This language reference manual details the features of MatrEL and how one can program in this exciting language.

2 Lexical conventions

In MatrEL there are six kinds of tokens: identifiers, keywords, constants, strings, expression operators, and other separators. A sequence of one or more separators is required in between tokens. Blanks, tabs, newlines, and comments are used as separators and are otherwise ignored by the compiler.

2.1 Comments

The string of characters that begins with # and ends with # is treated as a comment.

2.2 Identifiers (Names)

An identifier is a sequence of letters and digits. The first character must be a letter. The underscore, "_", symbol may be used as part of an identifier. Identifiers are case sensitive; uppercase and lowercase letters are considered different.

2.3 Keywords

The following identifiers my only be used as keywords:
int
double
matrix
cell
string
return
if
elseif
else
while
blank
and
or
equal
notEqual
Boolean
true
false
getInput

2.4 Constants

There are two types of constants in MatrEL:

2.4.1 Integer constants

An integer is a sequence of numbers 0-9, but the first digit cannot be 0.

2.4.2 Strings

A string is a sequence of characters enclosed in double quotes ‘ “ ‘.

3 What’s in a Name?

MatrEL interprets an identifier based on its storage class and its type. The storage class determines the location and lifetime of the storage associated with an identifier while the type determines the meaning of the values found in the identifier’s storage.

The two declarable storage classes in MatrEL are automatic and external. Automatic identifiers are local to each instantiation of a function and are discarded upon function exit. External identifiers, on the other hand, exist independently of functions.

MatrEL supports two primary types of objects:

Characters: letters a-z and A-Z
Integers: sequence of numbers 0-9

In addition to the primary types MatrEL also has the following derived types:

cell: a row,column value that corresponds to an entry in a matrix
matrix: a two-dimensional array of cells
string: a sequence of characters
functions

4 Conversion

This section explains how operand conversion occurs in MatrEL.

4.1 Integers and Strings

An integer may be converted to a string representation of itself. Likewise, a string may be converted to an integer given that the string is a string representation of a sequence of integers.
The example,
string a = “5”
int b = a
results in b = 5.

5 Expressions

Expressions may be grouped into sub-expressions by surrounding the sub-expression in curly braces {expression}.

5.1 Primary expressions

Primary expressions involving function calls group left to right.

5.1.1 identifier

An identifier is a primary expression so long as it is properly declared

5.1.2 string

A string is a primary expression consisting of alphabetic characters.

5.1.3 {expression}

An expression inside curly braces is an expression whose type is the same as the expression without curly braces.

5.2 Unary operators

Expressions with unary operators group right to left.

5.2.1 -expression

The result is the negative of the expression and has the same type. The type of the expression must be an integer.

5.2.2 rowCount myMatrix

rowCount applied to a matrix returns the number of rows in that matrix.

5.2.3 columnCount myMatrix

columnCount applied to a matrix returns the number of columns in that matrix.

5.3 Multiplicative operators

The multiplication operator * group left-to-right.

5.3.1 expression * expression

The binary * operator indicates multiplication. Both expressions must be integers.

5.4 Additive operators

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

5.4.1 expression + expression

The result is the sum of the expressions. Both expressions must be integers.

5.4.2 expression – expression

The result is the difference of the expressions. Both expressions must be integers.

5.5 Relational operators

The relational operators group left-to-right:

expression < expression	less than
expression > expression	greater than
expression <= expression	less than or equal to
expression >= expression	greater than or equal to

Both expressions must be integers.

5.6 Equality operators

equal expression expression	equal to
notEqual expression expression	not equal to

The expressions being compared must be of the same type. Within a comparison, the expression types may be boolean, integer, or string.

5.7 expression and expression

The and operator groups left-to-right. Both expressions must be boolean.

5.8 expression or expression

The or operator groups left-to-right. Both expressions must be boolean.

6 Declarations

Declarations are used to give a type and value to an identifier. They have the form:
typeSpecifier identifier = value

6.1 Type-specifiers

The type-specifiers are:

- int
- boolean
- string
- matrix
- cell

6.2 int declarators

int declarations have the form:
int identifier = value where value is a sequence of numbers 0-9.

6.3 boolean declarators

boolean declarations have the form:
boolean identifier = value where value is either true or false.

6.4 string declarators

string declarations have the form:
string identifier = “value” where value is a sequence of alphabet characters, including _ underscore.

6.5 matrix declarators

matrix declarations have the form:
matrix identifier = value where value is an integer and specifies the size of the square matrix. The matrix will have value number of rows and value number of columns.

6.6 cell declarators

cell declarations have the form:
cell identifier = val1,val2 where val1 and val2 are integers.

7 Statements

Statements are executed in sequence.

7.1 Expression statement

Expressions have the form
expression
and are typically assignments or function calls.

7.2 Compound statement

Statements can be executed in order by combining them into a compound statement which puts curly braces around the list of statements:
compound statement:
{statement-list}
statement-list:
statement
…
statement

7.3 Conditional statement

The three forms of the conditional statement are:

```
if {expression} statement

if {expression} statement
else statement

if {expression} statement
if-else {expression} statement
…
if-else {expression} statement
else statement
```

For each statement the expression must evaluate to a boolean. The statement is executed if the expression evaluates to true. If neither of the if-expressions evaluate to true, the else statement will be executed.

7.4 While statement

The while statement has the form:

while {expression} statement

The expression evaluates to a boolean. The statement is repeatedly executed while the expression evaluates to true.

7.5 Return statement

A function returns to its caller by means of the return statement, which has one of the forms:

return	no value is returned
return {expression}	the value of the expression is returned

8 Scope rules

In MatrEL, we must consider lexical scope which is the area of the program in which an identifier is accessible.

8.1 Lexical scope

There are two types of lexical scope, local and global. Identifiers declared within a function are local only to that function and may not be used otherwise. Global identifiers which are declared outside any and all functions may be used anywhere in the program.

9 Types revisited

This section summaries the operations that can be performed on objects of certain types.

9.1 Functions

Functions have the form:

functionReturnType functionName {parameter-list}
{function-body}

The function return type can be integer, boolean, matrix, cell, string or empty if the function will not be returning an object. The functionName is a valid identifier. The parameter-list will be of the form {type iden1, …, type iden2}. The function-body is an expression that evaluates to and returns the same type as functionReturnType.

9.2 Matrices

A matrix can be set the following ways:

myMatrix[myRow][myColumn] = “a”	Sets the matrix cell at row = myRow and column = myColumn to the string a. The cell value must be a string.

myMatrix[][] = value	Sets each cell value to value. Value may be a string or the keyword blank to be set to a blank string.
myMatrix[][myColumn] = value	Sets all of the cells in matrix column = myColumn to value.
myMatrix[myRow][] = value	Sets all of the cells in matrix row = myRow to value.
myMatrix[/] = value	Sets all of the cells in matrix diagonal (bottom left to top right diagonal) to value.
myMatrix[\] = value	Sets all of the cells in matrix diagonal (top left to bottom right diagonal) to value.
myMatrix[cellPos] = value	Sets the cell at location cellPos (integer, integer) in matrix to value.

In each of the above cases, the row and column values must be an integer.

A matrix cell value can be accessed the following ways:

myMatrix[rowNum][columnNum]	Returns the string value located at cell position rowNum,columnNum in myMatrix

Matrix values can be queried using the following keywords:

eachCellContains myMatrix value	Returns true or false whether or not each cell value in myMatrix equals value
anyCellContains myMatrix value	Returns true or false whether or not one or more cell's value in myMatrix equals value

9.3 Cells

Cells have the form:

cell myCell = rowNum,colNum

where rowNum and colNum correspond to a row and column position in a matrix.

Row and column values can be extracted from a cell by using:

myCell:row	returns the row number
myCell:column	returns the column number

10 Formated Output

The following output functions are made available in MatrEL:

printOut someString	prints the string someString to the console
printMatrix myMatrix	"pretty prints" the matrix myMatrix to the console

11 Example

The following example uses MatrEL to implement Tic-Tac-Toe.

```
# game Tic-Tac-Toe #

# initialize gameboard to a 3x3 matrix and set all cell entries to empty #
matrix gameboard = 3
gameboard[][] = blank

# create the three-in-a-row winning conditions #
boolean threeInARow {matrix m, cell pos, string value}
{
        if eachCellContains m[pos:row][] value
        { return true }
        elseif eachCellContains m[][pos:column] value
        { return true }
        elseif eachCellContains m[\] value
        { return true }
        elseif eachCellContains m[/] value
        { return true }
        else
        { return false}
}

# game loop #
boolean gameOver = false
string userInput = empty
string value = empty
cell userPos

while –gameOver
{
        printout “Enter selection as row,column:”
        getInput stdin userPos
        printout “Enter position value:”
        getInput stdin value
        gameboard[userPos] = value

        if threeInARow gameboard userPos value
                { printout “Player <value> wins!” }
        elseif -{anyCellContains gameboard empty}
                { printout “Game tied!  No more moves left!”

        gameOver = true
}
printMatrix gameboard
```