Zachary Craig

Final MyPI Project: Basic Linear Algebra Interface Writeup

Basic Declaration Grammar

Matrix declaration:

```
#semicolons indicate new rows. Commas indicate new columns.
Var matrix = [1.0, 0.0, 0.0; 1.0, 0.0, 0.0;1.0]
#Matrix entries are parsed as expressions
Var matrix = [1.0, 1.0 + 2.0 + 4.0 / 2, 3.0]
```

Assignment

```
A = B #assigns elementwise
```

Basic Arithmetic Grammar

```
#Adds matrices A & B element-wise
A + B
#Multiplies matrices A & B
A * B
#Multiplies matrices A & B element-wise
A .* B
#Multiplies matrices A & B using matrix multiplication
A * B
# Divides matrices A & B element-wise
A ./ B
# Subtracts Matrix A from Matrix B
A - B
#Exponentiates matrix A
A^2
#Exponentiates matrix A element-wise
A.^2
#Transposes Matrix A
~A
#Multiplies matrix by double or integer constant C
C * A, or C* A
#Mods each element in matrix A by B.
A % b
```

Built-in Functions

```
#returns R x C matrix of whose elements are A.
Fun matrix m_singleton(A:double,R:integer,C:integer)
#prints matrix with additional endlines
```

Zachary Craig

Fun nil m_print(A: Matrix)
#gets matrix entry from matrix
Fun double m_get(A: matrix, x:int,y:int)

How it was actually designed:

1.) First, I created tokens the following additional tokens:
{MATRIX_TYPE,"MATRIX_TYPE"}, {R_BRACKET,"R_BRACKET"},
{L_BRACKET,"L_BRACKET"},{SEMICOLON,"SEMICOLON"},{MATRIX_VAL,"MATRIX_VAL"},
//Dot operations
{DOT_MULTIPLY,"DOT_MULTIPLY"},
{DOT_DIVIDE,"DOT_DIVIDE"},{DOT_EXPO,"DOT_EXPO"}, {EXPO,"EXPO"}, {TRANSPOSE, "TRANSPOSE"}
};

2.) I then added cases to return these tokens in the lexer.

3.) I then built two new AST classes stemming off of RValue:

```
class MatrixValue : public RValue
{
public:
  Token first_bracket;
  vector<vector<Expr*>> M;



  // visitor access
  Token first_token() {return first_bracket;}
  void accept(Visitor& v) {v.visit(*this);}
};
```

```
class TransposedRValue : public RValue
{
public:
  Expr* expr = nullptr;
  ~TransposedRValue() {delete expr;}


  // visitor access
 Token first_token() {return expr->first->first_token();}
  void accept(Visitor& v) {v.visit(*this);}
};
```

4.) I then parsed matrix values and added their data to the AST.
5.) I then built a visitor for matrix value and transposed matrix value in the printer, type-checker, and interpreter.
6.) I then established simple rules in the type checker: i.e.
    a.) Matrix + Matrix-> matrix
    b.) Matrix - Matrix -> matrix
    c.) Matrix * (double || int)->matrix
    d.) Matrix % int->matrix
    e.) ect…...
7.) I then built a new matrix representation in DataObject.
8.) I evaluated the expressions in my interpreter's matrix visitor and stored each matrix into curr_val.
9.) Next, I built all the arithmetic cases for matrices in the Expr visitor of the interpreter.
10.) Lastly, I created new built-in functions for common linear algebra applications.

What was left out:

I left out row reduction, combining matrices, and span because they proved to be difficult to implement. I also could have made accessing individual values in the matrix a bit easier. Given more time, I would add these along with other useful linear algebra tools.

11.)

Challenges:

This project was time-consuming but not all that difficult given that we had already been through each step of creating a new programming language. One of the greatest challenges was figuring out how to represent a matrix and where to place its representation in the AST. Navigating your code to make DataObject Represent a matrix value was also quite difficult, since we had treated the DataObject class as a black box. There really is so much more to add to this extension. I'm sure running through a linear algebra textbook would generate many more additions to this extension. I hope it's helpful!

Zachary Craig