maTricks -- Language Specification

Introduction

Welcome to the world of matrices .We are going to bring you into the world where we see everything as a Matrix. **maTricks** is coined by our motivation of Matrices.

Main goals of the language

This language is designed for those work on the matrices integrated topics. ***maTricks main focus on expressiveness and performance of matrix operations***.We will make most of the matrix operations built in. Users working on matrices can create cool stuff from it.

1. Simple and *Expressive*
2. Matrix Operations
3. Performance : Least Time and easy computation of operations
4. *Familar*

Main language features

1. Primitive types
2. Expressions
3. Control flow
4. Functions
5. User defined types

Primitive Types

Type defines the set of values an object can take.
The main objects in our language are matrices. This is a generalisation which comes from a view that every thing can be represented by matrices which means any information can be encoded in terms of matrices and scalars. The basic data types available in our language are:

> *matrixInt*: denoted by keyword ***mInt***
>
> *matrixFloat*: denoted by keyword ***mFloat***
>
> *matrixString*: denoted by keyword ***mString***
>
> *Scalar*: denoted by keyword ***scalar*** *(scalar represents a floating point number.)*
>
> *Index*: denoted by keyword ***index*** *(index is especially used for indices in iterators. It takes only positive integers.)*

EXPRESSIONS

Expressions are key part of any language. A program can be viewed as a sentence which contains expressions. Thats how important they are.
Lets look at some of the valid expressions in the language. The following operators add expressive power.

1. " * " Is used for multiplication of 2 matrices.
   For instance A*B denoted multiplication of matrix A with B. First checks the feasibility of multiplication (number of coloums in A should be equal to the number of rows in B), if fails, outputs the error. Both A and B cannot be of mString type. Multiplication of mInt, mString returs a mString matrix where each element of input mString matrix is repeated corresponding element of mInt matrix number of times. If one of A,B is of mString type, then the other one should be mInt or mBool type.

2. " + " Is used to add two matrices or two scalars. Type checking is done here.
   For instance a+ B is invalid if a is scalar and B is matrix

3. " ** " Is used go multiply a scalar with a matrix
   For instance a**B means multiplication of scalar a with matrix B

4. " // " Is used go divide a scalar with a matrix
   For instance B//a means divide of scalar a with matrix B

5. " - " is used to subract two matrices or two scalars .Type checking is done here also.
   For instance a-B where a is scalar and B is matrix is not valid. Other instance which is not valid is the order check fail. i.e A-B is not valid if A and B are not of same order.

6. " $ " is used to get the sum of all the elements in the required matrix.
   For instance, "$A" returns the sum of all the elemtns in the matrix A. If the matrix A is empty, returns 0.

7. " ’ " is used to evaluate the transpose of the matrix.
   Usage: "A’" returns the matrix that is transpose of matrix A.

8. "#" is used to evalute the inverse of the matrix.
   Usage: "A#" returns the inverse matrix of A. But firstly it checks if the matrix is inversible, if not, prints error message.

CONDITIONALITY OPERATORS: (relational operators)

All the operations return 1 if true else 0 if false .

1. "=="
   Checks the dimensions of the matrices, if not same returns 'false', if same, then checks each and every element in both the matrices, if any corresponding element is found different, returns 'false' else goes on to check all the elements, if all are same, returns 'true'.
2. "!="
   Checks the dimensions of the matrices, if not same returns 'true', if same, then checks each and every element in both the matrices, if any corresponding element is found different, returns 'true' else goes

on to check all the elements, if all are same, returns 'false'.
Usage: A!=B, returns "true", if A is not equal to B, else returns 'false'.

3. "<"
   Checks the first element of each matrix and compares them, if same moves to the next element of same row, if there is no next element in the same row, then moves to the next row of that praticular matrix and compares it with the next element in other matrix. The matrix in which the first greater element is found, is greater than the other matrix.
   Usage: A<B, returns 'true', if A is less than B, else returns 'false'.
4. ">"
   Checks the first element of each matrix and compares them, if same moves to the next element of same row, if there is no next element in the same row, then moves to the next row of that praticular matrix and compares it with the next element in other matrix. The matrix in which the first smaller element is found, is smaller than the other matrix.
   Usage: A>B, returns "true", if A is greater than B, else returns 'false'.
5. "<="
   Usage: A<=B, returns "true", if A is less than or equal to B, else returns 'false'.
6. ">="
   Usage: A>=B, returns "true", if A is greater than or equal to B, else returns 'false'.
   Remark:- relation operators return 1 if they are true 0 if they are false.

BITWISE OPERATIONS:

1. '|' (Element wise OR operator)
   Usage: A|B, first checks the dimensions of A,B, if not same, sends the error message to the user, else, does the OR operation for the each corresponding elements of matrices A,B.
2. '||'
   Usage: A||b, does OR opeartion of all elements of the matrix A with given scalar b. If 'b' isn't a scalar then sends error message to the user.
3. '&'
   Usage: A&B, first checks the dimensions of A,B, if not same, sends the error message to the user, else, does the AND operation for the each corresponding elements of matrices A,B.
4. '&&'
   Usage: A&&b, does AND opeartion of all elements of the matrix A with given scalar b. If 'b' isn't a scalar then sends error message to the user.
5. '!'
   Usage: !A, does the bitwise NOT operation for all the elements in the matrix.
6. '~'
   Usage: A~B, firstly cheks the order of the matrices A,B, if not same, sends the error message to the user, if same, does the bitwise XOR operation on the corresponding elements of the matrices A,B.
7. ' ~~ '
   Usage: A~~ b, firstly cheks if b is scalar or not, if not scalar, sends the error message to the user, if scalar, does the bitwise XOR operation on each element of the matrix A with b.

CONTROL FLOW

control flow (or flow of control) is the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated.

***If-Else* statement**

if, else are the keywords and { is used for opening the statement and } used for closing the statement

```
if(index){
    statements
}else if(index){
    statements
}else{
    statements
}
```

Remark:- Boolean is not a primitive data type in our language. So booleans must be encoded in the form of scalars or indices.(> 0) is evaluated as true.

Loops

for is the keyword and statements are present in the { statements }
Most programming languages have constructions for repeating a loop a certain number of times. In most cases counting can go downwards instead of upwards and step sizes other than 1 can be used. (intialization;condition;function of the variables)

```
for(index i=0;i<INDEXtype ;i++)
{

}
```

Functions

Def is the keyword used for defining any function .arguments are present in the function brackets return type is present in the brackets .Return tyrpe is given at the function

```
Def functionName(Args) =>returntype
{
  statements
}

Def f(mInt x,mInt y)=>mInt
{
     statements
     return expression;
}
```

```
//function calling:
f(x,y)
```

User Defined Types

User-defined functions are functions that you use to organize your code in the body of a policy.mStruct is matrix struct which means it is matrix of struct element.

```
mStruct Dtype
{
    mInt Aa;
};
Dtype Name(2,1);
```

Memory Model

All the objects once created are in the memory untill program ends or explicilty cleared i.e no garbage collector.

Examples

Code 1: Operations between variables

```
mInt A(1,1) = 2;
mInt B(1,1) = 4;
mInt C(3,3) = (){return ROW+COLUMN+1;} #function of ROW ,COLUMN WHERE  THEY ARE
KEYWORDS for intialization
mInt Sum(1,1) = A+B   ;     #Matrix Addition
mInt Subtract(1,1) = A-B  ; #Matrix Subtraction
mInt product (1,1) = A*B ;  #Matrix Multiplication
scalar a=4;
mInt division(1,1) = A//a  ;
#Matrix division by scalar
```

Code 2: Student details

```
mStruct Person
{
    mString Name;
    mString Age;
    mString Gender;
```

```
    mString DOB;
};
Person persons(1,2);
/*  btech column     mtech column  (these are the two columns persons are the
elements of rows)*/
persons[0][0].Name = “Ramu”;    /*btech person*/
persons[0][0].Age = “25”;
persons[0][0].Gender = “Male”;
persons[0][0].DOB  = “13/01/1996” ;

persons[0][1].Name = “Laxman”;  /*mtech person*/
persons[0][1].Age = “27”;
persons[0][1].Gender = “Male”;
persons[0][1].DOB  = “23/09/1994”;
```

Code3:

```
mInt persons(1,2)=1,0;  //wrong  to be [1,0]
mInt p1(2,2)=[0,1;1,0]; //right
mInt p2(1,1)=[1];       //is same as  only for (1,1)
mInt p3(1,1)=1;
```

Code4:

```
/*This example illustrates the concept of typeCasting,
 * Inversion, operator Precedence and associativity*/

//Solving for system of linear equations Ax=B
Def SolveEquations(mInt A,mInt B)=> mFloat{
    //Inverse operator left associativity
    //Precedence of Inverse is greater than Multiplication
    //Here before the Inversion of matrix it should be
    //typecasted to Float as the inversion of matrix can
    //lead to loss of precesion
    mFloat X= B*(mFloat)A#;
    return X;
}
```

Difference between maTricks and Reference Languages:

- There is no auto type casting from one data type to another but the data type matrix you choose is the final and for example: If you try to add float values to MATRIXINT then there will be a compile error so in our language you add data only of the type of MATRIX you have declared.

- Our language does not support OOPs which is a feature of C++ and since we are not using Object Oriented Programming there is no need for Classes which also removes the features of Inheritance and Polymorphism.
- Our language does not support any kind of explicict operator overloading which is a feature of C++ as it is not required in our language as we have created all the required functions with user friendly keywords.
- Like in C/C++ there are no primitive data types like int,float,char,bool and arrays but the only datatypes in our language are the matrices having different data types.
- There are no goto statements as they effect readability of code and mostly not used in any language and we have only break and continue statements which are extensively used and also there is absence of the do-while loop.
- Any matrix you declare in our language has a global scope and not block scope which will make it easy and user friendly.
- There is no need of pointers as there is no need of referencing since all changes in the matrices can be done as they have a global scope which make it easy since there is no hectic of dereferencing it to change values.

Compiling and Execution

- If there is chance of range/size inference without ambiguity range is infered automicatially without any specification
- Compile Errors
    - Type checking is performed during the phase of compiling and coresponding errors will be reported
        - Examples: In terms of sizes of matrices like addition of matrices expects the meatrices of same size and product of two matrices expects the number of coloums of first matrix is same as number of rows of second matrix.
    - All the contents of the elements of the matrix should be of same type i.e the Matrix is homogenous container.
- Trying to access the elements that are out of range of matrices results in run time errors.

Performance

- All the parts of the code which potentially can be parellelised will be identified and parellelised.
- Examples: For a large enough matrices operations like addition and matrix multiplication can be parellelised, So that time of execution outweighs the time of object creation.

Possible future extensions

- This language can be made to fit into the popular languages like C/C++/Python. So, that current context can be utilised.
- The embedded code written in host language can be translated into the host language at a preprocessor stage to get the desired effect.

- Alternatively, a separate compiler toolchain extended from the host language can analyse the AST built and then generation of machine code for the host language can be handled by host compiler and code generation of this language can be written.
