Angela_Z Final Report

2014/12/17
W4115 Programing Language and Translators

Fei Liu (fl2312)
Taikun Liu (tl2582)
Jiayi Yan (jy2677)
Mengdi Zhang (mz2472)

1.Introduction to Angela_Z

1.1Background

The idea of matrix dating back to the ancient Chinese math book “Nine Chapters of the Mathematical arts” between 300 BC and AD 200. Today, we made this modern attempt to celebrate once more for the wisdom of our ancestor and help our gorgeous friends working in investing banks a little bit.

Angela is a nickname for Angel and Z stands for Zen. Zen is a kind of Buddhism belief, which calls on people to sit there doing nothing, like a sleeping angel/beauty. We hope people can take advantage of our language and take a break with their life with ease.

1.2Related Work

Matlab has similar implementation. However, their approach of handling strings in a cell and regard everything as a matrix object is not intuitive to everyone especially our friends in financial industry. People, especially people working in financial district, prefer a way of doing the business in a flat and simple fashion. We take advantage of the stringification of any data structure related to our implementation and let the user pass in whatever data type they care to use as a String and they themselves convert back to the data type they want to use. In this case, complicated extension and inheritance is not necessary and everything falls into argument passing and processing. In this sense, not everyone is bored with the Matrix operation and they can actually play with any data type they care to use hopefully with ease and retain their instinct with primary data type from their intro programming classes, while enabling complex structures and String manipulation that they bother using.

Operators are intuitive as we keep tattooing them through adulthood or earlier. However, handy language like Java does not allow overloading of operators and the C++ way is complicated. Is there a way that differentiates the operation between Matrixes, primary types, or between Matrix and Primary type? Yes, Angela zens.

Quantitative finance is sophisticated and challenging because of our ignorance of applying certain knowledge. How to help new financial employees hook up with their job quicker and easier? We introduce an instance of utilizing our language to price options based on single one entry and based on a matrix input.

1.3 Goals of Angela_Z

As always, "Faster, Higher, Stronger". No kidding, Faster as the complicated Matrix operations are tokened like in a linear algebra textbook; Stronger as the functionality and logic can be much stronger; and Taller as any user of this language stands on the shoulder of gigantic 4 cool coders who knows and solve problems for them before they even started using it. As always, "Go and beyond". The extensibility of a language is very important. How to extend the usage of a language to a foreign field is important. We offer a way of customizing the language to a certain field by extending the build-in Structure type and then specify what is required for certain type of computation and further define user-defined functions that handles those pass-in's gracefully behind the scene and the end user of those extended functionalities can then benefit from it and rest in ZEN!

2. Tutorial

2.1 Sample Code

2.1.1 "Live with Matrix"

```
Int i;
Boolean b;

Matrix main2(Int argc, String argv) {
Matrix m3(2,2);
Matrix m(2,2);
m[0][0]=1;
m[0][1]=2;
m[1][0]=3;
m[1][1]=4;
m3 = (((m +. m') *. m~) *.. 4)+.. m^;
return m3;
}

Void main(Int argc2, String m) {
```

```
  Matrix result(2,2);
  result=main2(0, "str");
  printM(result);
}
```

This demo illustrate the usage of the assignment of matrix elements and usage of matrix operators and then how to nice-print the result of the computation.
First, access the elements of the matrix just like in a two dimensional array and we handle the internal structure for you.
Second, use the pre-defined operators to apply the build-in Matrix calculations. The trick is that the plane "+"/"-" are for the primary type, the "+.", "-.", "*.", and "/." are matrix-wise operations, and the "+..", "-..", "*..", and "/.." are the mixture of the matrix and a primary type Float or Int. However, the order of the operands matters. Float or Int has to be after the Matrix in such operation. We can make it more free-stylish, but we want to coerce ordering, which is helpful in writing maintainable and easy-to-understand code, which is also what Ocaml did to us

2.1.2 "Live with Struct"

```
Int i;
Boolean b;

Structure main2(Int argc, String argv) {
  Structure s={a="1", b= toString(argc)};
  i=toInt(s -> a);
  return s;
}

Void main(Int argc2, String m) {
  Structure result={};
  result=main2(0, "str");
  print(result);
}
```

This demo illustrate the usage of Struct type. This is a type to pass useful information only in String type. We can easily convert any type to String pass it and then convert it back. Instead of tracking the type, we give the user the opportunity to pass in anything and it will work just fine as long as a user knows how to retrieve it. This approach push the responsibility to the user and the user can thereby choose to implement anything from their own application and would not have nay compiler error. Also, compared with the Java extension and inheritance approach. Our way has little overhead to be familiar with the OOP design pattern and the rules of extension. They however assume the responsibility to understand what they actually passed into the function and convert it back using the right function.

2.1.3 "Application in Option"

```
Float i;
{*
Here is the comment you don't care about!
*}
Option main2(Int argc, String argv) {
  Option s={strike="100.0", stock= "150.0", interestRate="0.1", period="1.0",
sigma="2.0", optionType="call"};
  i=toFloat(s -> strike);
  return s;
}
Matrix main3(Int a) {
  Matrix strike(1,2);
  strike[0][0]=10;
  strike[0][1]=20;
  Matrix stock(1,2);
  stock[0][0]=15;
  stock[0][1]=25;
  Matrix interestRate(1,2);
  interestRate[0][0]=0.4;
  interestRate[0][1]=0.1;
  Matrix period(1,2);
  period[0][0]=3;
  period[0][1]=4;
  Matrix sigma(1,2);
  sigma[0][0]=0.1;
  sigma[0][1]=0.2;

  Matrix s(0,0);
```

```
  s= priceM(strike,stock,interestRate,period,sigma);
  return s;
}
Void main(Int argc2, String m) {
  Option result={};
  result=main2(0, "str");
  Float d;
  d=price(result);
  print(d);
  Matrix result1(0,0);
  result1=main3(0);
  print(result1);
}
```

This demo illustrates the usage of extending the Structure into customized approach and get adapted for a certain application.
The financial world has many ready to use closed-form formula and the users actually don't need to repeatedly do things over and over again. By extending just the Structure, we allow the application to be fully application based by allowing only certain parameters to pass in and customized methods to be called. This approach is enforced by allowing the operation to operate on top of Matrix and the computation can be easily loop free and whatever optimization and underground work can be possible.
In this demo, we create Matrixes filled with parameters for a portfolio of options and for each of them we want to calculate a price. The parameters are passed in as a matrix and the priceM() method takes those matrix and compute the result and then the result is printed.

2.2 File Organization

Angela-z's source code file has an extension ".az", such as "code1.az". The first type of compile is to do the type checking. Type checking does not yield any new files but throw exceptions when it finds errors. After type checking, you can use another command line to generate java codes files.
The java codes files' structure are:
User can then run the java file within the right folder.

2.3 Compiling and Running

2.3.1 Generate the code

```
# to make the depository clean
$ make clean
# making the compiler
$ make
# the output will be saved in java/Output.java
$./angelaZ < <your_source_code>
# Copy everything in the java folder and put it into an eclipse Java-project
environment to run.
Open the Output.java and run from it.
```

2.3.2 Test script

(PLEASE NOTE that repeating “make clean” is important at each step because the target for Makefile is different and the dependency is different.)

```
#### test a single file
# to make the depository clean
$ make clean
# making the compiler
$ make
# the output will be printed to the console
$./angelaZTest < <your_source_code>
#### test a single file
$ make clean
# compiles
$ make
# the output will be printed to the console
$ ./testall.sh
```

2.3.3 Type checking

```
# to make the depository clean
$ make clean
# making the compiler
$ make
```

$./angelaZ -s < <your_source_code>

If semantic error happens, it will throw an exception indicating the error, such as “cannot use keyword xxx as variable name ”. If no error happens, it will show that "Semantic analysis completed successfully.". This procedure does not output any file.

3. Reference Manual

3.1 Language definition

3.1.1 Usage

Angela-Z is a language designed for matrix computing with build in module specializing on facilitating option pricings. It can also be used as a general-purpose programming language with similar modifications made to facilitate option pricing. (In other words, option specialization is just a demo of usage of matrix computation. The true application can be as wild as human imagination.)

3.1.2 What special feature do we have?

We support the built-in Matrix operators, Matrix I/O, and Option pricing features. Matrix-wise operations such as Plus, Minus, Multiply, Division, Transpose, Inversion, and Determinant are all supported. Facilitating functions for nice matrix printing and element accessing are also included.
Option as an concrete application of using the toolbox above is provided for pricing with Black-Shores formula. The language should be able to shelter the user from intensive loops and manipulation of matrices or collection of option pricing data.

3.2 Lexical Conventions

3.2.1 Comments

Comments are delimited by {* and *}, like

```
{* This a single-line comment.*}
{* This
is a
block
comment.*}
```

We currently only support block comments and no single-line comment syntax like // in c++.

Nested comments is not supported as in C++. I don't think it worth the effort.

3.2.2 Identifiers

An identifier is a combination of a series of characters, which includes letters, digits as well as underscore'_'. Identifiers can be used as name for variables and functions. The identifiers are case sensitive which means that 'Foo' is not the same as 'foo'. And it can only start with letters.
There are several key words reserved from our system, detaily listed in section 2.3. Those keywords cannot be used as identifiers.

3.2.3 Keywords

The following are a list of reserved keywords in the language and can not be used as variable or function names.

if
else
for
while
return
Boolean,true, false
Matrix
Structure
Option
Int
Float
String
Void

3.2.4 Literals Types

Literal primitive types

The following table contains some examples of literal primitive types.

Int	0 1 -30 99
Float	0.0 -0.002 2e-3 99.0
String	“Hello World”
Boolean	true false

Int

An integer consists of a sequence of digits and no decimal point and can start with a “-” to indicate that it is negative. A positive integer has no “+” sign. Integers are represented only in decimal notation.

Float

A float can be represented in two forms. One consists of an integer part and a decimal part. The second form also contains an integer part, as well as an e or E, an integer exponent.

String

A string is enclosed in double quotation, for example "Hello World". This data type can store a string of potentially unlimited length and does not have an upper bound limit. The length is only limited by the amount of computing resources (e.g. memory) available.

Boolean

A boolean is either true or false. It has no other value.

Literal collection type

Matrix

This type only supports two dimensional matrix. A matrix data type consists of rows and columns of only floats and could not be less than 1. The elements in the matrix can be accessed by its index starting from 0, for example m[0][1] can access the element in the first row and second column of matrix m. This can store a matrix of potentially unlimited length along each dimension and does not have an upper bound limit. The size of this type is only limited by computing resources available.

3.2.5 Object types

Structure

This type is like the struct in C language. A structure can take an unlimited amount of fields or elements. But the fields can only be string type variables. The structure can not be declared without initialization. and all fields must be declared at that time. Otherwise, the structure has no fields at all. The operator -> is used to access its member fields. We do not support adding or deleting fields to an structure. However one can change the value of fields within a structure.

Option

This type is an extension of Structure Type. It has some built-in functions and preset (key, value) pairs. Like structures, the fields of options can only be string type variables. An option can not be declared without initialization. The operator -> is used to access its member fields.

3.2.6 Operators

An operator is an evaluation performed on one or more operands. Each data type has its own set of operators.

Operators for Int, Float

Symbol	Explanation
+	Performs addition
-	Performs minus
*	Performs multiplication

/	Performs division
=	Performs assignment

Operators for String

Symbol	Explanation
=	Performs assignment
+	Performs concatenation

Operators for Matrix

Symbol	Explanation
+.	Performs elementwise addition between two matrixes
-.	Performs elementwise minus between two matrixes
*.	Performs elementwise multiplication between two matrixes
/.	Performs elementwise division between two matrixes, basically it is same with matrix *. inversion(matrix)
=	Performs assignment
(,)	Performs initialization
[ ][ ]	Performs index accessing

Operators for Matrix and Int/Float

Symbol	Explanation
+..	Matrix+..Int or Float, add Int or Float to all Matrix elements
-..	Matrix-..Int or Float, subtract Int or Float from all Matrix elements
*..	Matrix*..Int or Float, multiple Matrix elements with Int or Float
/..	Matrix/..Int or Float, divide all Matrix elements by Int or Float

Operators for Option, Structure

Symbol	**Explanation**
=	Performs assignment
{ }	Performs field initialization
->	Performs field accessing in the object

Operators for Boolean

Symbol	**Explanation**
&&	Performs AND operation of two Boolean expressions
||	Performs OR operation of two Boolean expressions

Equality Operators

Symbol	**Explanation**
==	Test whether two expressions are equal
!=	Test whether two expressions are different

Relational Operators for only Int and Float

Symbol	**Explanation**
<=	Test whether left expression is smaller or equal to right expression
<	Test whether left expression is smaller than right expression
>=	Test whether left expression is greater or equal to right expression
>	Test whether left expression is greater than right expression

3.2.7 Punctuators

A punctuator is a symbol that does not specify a specific operation to be performed. It has a syntactic meaning to compiler and is primarily used in formatting code. A punctuator is one of the symbols below:

Symbol	Example
;	Statement terminator
{}	Block of Statements
()	operation precedence forcing symbol

3.3 Program Structure

3.3.1 Basic structure

An Angela_Z program consists of variable declaration statements and function declaration statements. A program has only one entry function which is the main function that takes an int and a string as its arguments. Variables declared outside main function are global variables. Global variables can not be assigned to any value outside the main function, i.e its initialization must take place inside main.

3.3.2 Scoping

A global variable is a variable declared outside function declarations. A global variable is accessible from all position in the file after declaration. A local variable is a variable declared inside a function declaration. A local variable's valid scoping range is only inside the blocks it’s declared in.

3.4 Type Conversions

Type conversion from Int to Float is allowed. When an operation, such as + - * / < <= > >=, is conducted between a Float and an Int, the Int will be explicitly converted to Float first and then do the operation. Or an Int is given where a Float is expected, the Int will also be converted to Float. Reverse conversion from Float to Int is not allowed. Conversion for other types are not allowed. For example, Boolean can not converted to Int or vise versa.

3.5 Expressions and Operators

3.5.1 Precedence and Associativity Rule

The following table is a list of operator precedence and associativity for operators. Operators on the same row are of the same precedence and the table is in the order from highest to lowest precedence. Parentheses can be used to force precedence in the language.
Initialization, i.e., Structure or Option initialization, e.g op{ foo = “1” , bar =” 2” } and Matrix initialization,e.g m (1, 2), only appears in expression which contains only itself, so there is no need to assign precedence order to initialization.

Operator Symbol	Operator Description	Associativity
-> [] [] ()	Structure or Option element accessing Matrix element accessing Function call	Left
-	negative sign	Non-associative
* / *. /. *.. /..	Times, Divide	Left
+ - +. -. +.. -..	Plus, Minus	Left
< <= > >=	Less than, Less than or equal to, Larger than, Larger than or equal to	Left
== !=	Equality, Inequality	Left
&&	Logical AND	Left
||	Logical OR	Left
=	Assignment	Right

3.5.2 Primary Expressions

The following are all considered to be primary expressions:

Identifiers: An identifier refers to either a variable or a function. Examples include x_1 and hilbert, but not 2nd.
Constants: A constant's type is defined by its form and value. See the table in Section 2.4.1 for examples of constants.
String literals: String literals are translated directly to Java strings by our compiler, and are treated accordingly.
Parenthesized expressions: A parenthesized expression's type and value are equal to those of the un- parenthesized expression. The presence of parentheses is used to identify an expression's precedence and its evaluation priority.

3.5.3 Function Calls

To be able to call a function, it must be declared and implemented before. Calling a function that hasn't been defined before causes an error.
To call a function, the syntax is: function_name(first argument, second argument,....). The function call returns the value of the data type defined as return type in the function declaration. Note that when calling the function, the arguments should be of the same order and the same data types as defined in function declaration. Otherwise, error occurs.

For example, if a function declaration is as follows:

```
Int diff(Int a, Int b)
{
    return a-b;
}
```

Valid function call is like:

```
diff(0, 1);
```

Invalid function calls are as follows:

```
diff(0, 1.2); {* second argument is of incorrect data type *}
diff(0); {* number of arguments is different from what is defined in declaration
*}
```

3.5.4 Additive Operators

The additive operators are binary operators with left-to-right associativity:

- Plus (+)
- Minus (-)
- Matrix Plus (+.)
- Matrix Minux (-.)
- Matrix Int/Float Plus (+..)
- Matrix Int/Float Minus (-..)

Types used with the first two additive operators are Int and Float. Type conversion from Int to Float applies here if plus or minus is done between an Int and a float. The result of the plus (+) operator is the sum of the operands. The result of the minus (-) operator is the difference between the operands.

Examples are as follows:

```
Int a;
Int b;
a = 3;
b = a + 2;
{* b = 5.*}
```

Types used with the middle two additive operators are Matrix. Matrix Plus (+.) operator performs entrywise summation of elements of two matrixes. Matrix Minus (-.) operator performs entrywise subtraction of elements of two matrixes. Note that Matrix before and after operators should of same size. All elements in the matrix are treated as Float.

Examples are as follows:

```
Matrix a(1,2);
a[0,0] = 1.0;
a[0,1] = 5.0;
Matrix b(1,2);
b[0,0] = 3.0;
b[0,1] = 3.0;
Matrix c(1,2);
```

```
c = a -. b;
{* c is a matrix of size 1*2: -2.0,2.0 *}
```

Types used with the last two additive operators are a Matrix and an Int/Float. Matrix Int/Float Plus (+..) operator performs entrywise summation of matrix elements and the number. Matrix Int/Float Minus (-.) operator performs entrywise subtraction of matrix elements and the number. Since matrix only contains data of Float type, Int will be explicitly converted to Float first.

Examples are as follows:

```
Matrix a(1,2);
a[0,0] = 1.0;
a[0,1] = 5;
Int b = 1;
Matrix c(1,2);
c = a +.. b;
{* c is a matrix of size 1*2: 2.0, 6.0 *}
```

1.1.1. Multiplicative Operators

The multiplicative operators are binary operators with left-to-right associativity:

- Times (*)
- Divide (/)
- Matrix Times (*.)
- Matrix Divide (/.)
- Matrix Int/Float Times (*..)
- Matrix Int/Float Divide (/..)

Types used with the first two multiplicative operators are Int and Float. Type conversion from Int to Float applies here if multiplication or division is done between an Int and a float. The multiplication operator (*) yields the result of multiplying the first operand by the second. The division operator (/) yields the result of dividing the first operand by the second. Division by 0 is undefined and not allowed.

Examples are as follows:

```
Float a;
Float b;
a = 1.1;
```

```
b = a * 0.2;
{* b = 2.2*}
```

Types used with the middle two multiplicative operators are Matrix. Matrix multiplication (*.) operator performs row-by-column multiplication of two matrixes. Matrix division (/.) operator performs row-by-column division of two matrixes. Note that number of columns in the Matrix before the operator equals the number of rows in the one after the operator. Dimension checking will be done and error will occur in case of dimension mismatch.

Examples are as follows:

```
Matrix a(1,2);
a[0][0] = 1.0;
a[0][1] = 2.0;
Matrix b(2,1);
b[0][0] = 3.0;
b[1][0] = 2.0;
Matrix c(1,1);
c = a *. b;
{* c is a matrix of size 1*1: 7.0 *}
```

Types used with the last two multiplicative operators are a Matrix and an Int/Float. Matrix Int/Float Multiplication (*..) operator performs entrywise multiplication of matrix elements and the number. Matrix Int/Float Division (/..) operator performs entrywise division of matrix elements and the number. Since matrix only contains data of Float type, Int will be explicitly converted to Float first.

Examples are as follows:

```
Matrix a(1,2);
a[0][0] = 1.0;
a[0][1] = 5.0;
Int b = 2;
Matrix c(1,2);
c = a /.. b;
{* c is a matrix of size 1*2: 0.5, 2.5 *}
```

3.5.5 Relational Operators

The relational operators are binary operators with left-to-right associativity:

- Less than (<)
- Greater than (>)
- Less than or equal to (<=)
- Greater than or equal to (>=)

Types used with the relational operators are Int and Float. They yield values of type Boolean. The value returned is false if the relationship in the expression is false; otherwise, the value returned is true. Relation comparison of an Int and a Float is supported and the Int will be converted to Float first.

Examples are as follows:

```
Float a;
Int b;
a = 1.1;
b = 2;
if ( a >= 1.1 )
    {…}
  if ( a < b )
      {…}
{*
    a >= 1.1 returns true
    a < b returns true
*}
```

3.5.6 Equality Operators

The equality operators are binary operators with left-to-right associativity:

- Equal to (==)
- Not equal to (!=)

Types used with the relational operators are Int and Float and String. The equal-to operator (==) returns true if both operands have the same value; otherwise, it returns false. The not-equal-to operator (!=) returns true if the operands do not have the same value; otherwise, it returns false. Equality comparison of different types is not supported.

Examples are as follows:

```
        Float a;
        String b;
        a = 1.1;
        b = “str”;
        if ( b == “st” )
    {…}
if ( a == b )
    {…}
        {*
            b == “st” returns false
            a == b gives an error since a is Float and b is String
        *}
```

3.5.7 Boolean Operators

The Boolean operators are binary operators with left-to-right associativity:

- AND (&&)
- OR (||)

Types used with the Boolean operators are Boolean expressions. The logical AND operator (&&) returns the Boolean value true if both operands are true and returns false otherwise. Does not support (1 && 1==1)

Examples are as follows:

```
        if ( true && false )
            {…}
        if ( 1 == 1 || false )
            {…}
        {*
            true && false returns false
            1 == 1 || false returns true since 1 == 1 is evaluated as true
        *}
```

3.5.8 Assignment Operators

The assignment operators are binary operators with right-to-left associativity:

- Assign (=)

Types used with the assignment operators are Int, Float, Boolean, String, Structure, Matrix, Option, Assignment stores the value of the second operand in the first operand. Assignment between different types are not allowed. Assignment can not be done within variable declaration. A variable has to be declared first then assign.

Examples are as follows:

```
Float a;
Float b;
a = 1.1;
b = a * 2.0;
{*
    a is assigned value of 1.1
    b is assigned value of 2.2
*}
```

3.5.9 Initialization Operators

The Initialization Operators have no associativity:

- Structure or Option initialization (({ foo="", bar = "" })
- Matrix initialization (<nrows>, <ncols>)

Types used with the first initialization operator are Structure and Option. It should be of format Type Id{key=value, key=value….}. And an initializer list must exist in the declaration statement of the variable. Note that trying to access a Structure or Option that hasn't been initialized causes error. Multiple fields are separated by a comma in the initializer list.

Examples are as follows:

```
Option a{id="5", country= "China"};
```

{ a is initialized as Option Object which has two elements: id("5") and country("China") *}*

Type used with the second initialization operator is Matrix. The first value in the parentheses is the number of rows and the second value is the number of columns. Both values have to be positive Int and use of other types or 0 or negative Int causes error. Elements in the matrix is initialized as random Float numbers. Note that trying to access a Structure or Option that hasn't been initialized causes error.

Examples are as follows:

```
Matrix a(1, 2);
{* a is initialized as Matrix Object with size 1*2 *}
```

3.5.10 Element Access Operators

The Element Access Operators:

- Structure or Option element Access (->)
- Matrix element Access ([ ][ ])

Types used with the first element access operator are Structure and Option. It returns the element with if the id indicated by the string after the operator. Before trying to access elements, existence of the element of id will be checked and error occurs if it doesn't exist.

Examples are as follows:

```
Option a{id="5", country= "China"};
if ( a->country == "China")
    {...}
{* a->country returns "China" and so a->country == "China is true *}
```

Types used with the second element access operator are Matrix. It returns the object in the index position indicated by the pair [row index][column index]. User can either get the value in the position or assign new value to the element. Row and column starts from 0. An index range check is performed before trying to access elements and error occurs if index is out of range.

Examples are as follows:

```
Matrix a(1,1);
  a[0][0] = 1.0
Float b;
b = a[0][0]
{* b = 1.0 *}
```

3.6 Declarations

3.6.1 Function Declarations

Functions consist of a function header and a function body. The function header contains return type, function name, and parameter list. The function name must be a valid identifier. The function body is enclosed in braces. For example:

```
Int foo( Int a, Int b) { … }
```

3.6.2 Variable Declarations

Variables are declared in the following way:

```
dataType    varName;
```

dataType could be int, float, string, boolean, void, matrix, structure, option. varName must be a valid identifier that is a combination of characters, and can only begin with letters.
Matrix, structure and option types have their own declaration convention. Matrix are declared with their dimensions specified in the parentheses and its columns and rows could not be less than 1. For structures and options, they are declared in the same way. Their member fields are enclosed in the curly braces and are separated by a comma. They must be initialized during declaration. See examples below.

The variables and constants could be initialized with a literal value, an arithmetic expression or be set equal to another variable or constant that has already been declared in this scope. The type of the value and the type of variable it has been assigned to must match with each other. The following are some examples of variable declaration and initialization.

```
Int a;
Int a = 3;
Int a = 2 + 3;
Int b = a;
Matrix m(1,2);
Structure s { a = "Hello", b = "World"};
Option o { a = "foo", b = "bar"};
```

3.7 Statements

3.7.1 Block

A block encloses a series of statements by braces.

```
{
stmt1;
stmt2;
stmt3;
}
```

3.7.2 Conditional Statement

There are two forms of conditional statement that consist of a "if" and an optional "else".

```
if ( boolean expression) { stmt1 }

if ( boolean expression)   stmt1
else stmt2
```

The boolean expression will be evaluated first, and if it is true then stmt1 will be executed, otherwise, in the second case, stmt2 will be executed.

3.7.3 For Loops

For loops are in the following format:

```
for   (expr1, b_expr, expr2) stmt
```

expr1 is evaluated only once during first iteration, the b_expr is a boolean expression and it is checked before each iteration, if it is true, then the stmt is executed. expr2 is executed in the end of each iteration. Both expr1 and expr2 have to be assignment expressions.

3.7.4 While Loops

While loops are in the following format:

```
while (b_expr) stmt
```

The b_expr is a boolean expression and it is evaluated before every iteration. If it is true then the statement stmt will be executed. The loop will stop when b_expr is evaluated false.

3.7.5 Return Statement

Return statement have two formats:

```
return stmt;
return;
```

Functions could contain a return statement, it could have another statement or just an empty return statement.

3.8 System Functions

3.8.1 print() Function

The print function is used to output a variable's value or a string literal. For example:

```
Int a = 3;
print (a);
{* 3 *}

print ("Hello World!");
{* Hello World! *}
```

3.8.2 printM() Function

Print the Matrix.For example:

```
Matrix m(1,1);
m[0][0]=4.5;
printM(m);
```

3.8.3 toInt() Function

Convert a String to a Int if the format is correct. For example:

```
String a="1";
print(toInt(1)+1);
```

3.8.4 toFloat() Function

Convert a String to a Float if the format is correct. For example:

```
String a="1.5";
print(toFloat(1)+1.0);
```

3.8.5 toBoolean() Function

Convert a String to a Boolean if the format is correct. For example:

```
String a="false";
print(toBoolean(1));
```

3.8.6 toString() Function

Convert any object with type other than Void into String. For example:

```
Matrix a(1,1);
print(toString(a));
Option b={myname="1"};
print(toString(b));
```

3.8.7 price() Function

Price for a well-defined option. For example:

```
    Option main2(Int argc, String argv) {
Option s={strike="100.0", stock= "150.0", interestRate="0.1", period="1.0", sigma="2.0",
            optionType="call"};
i=toFloat(s -> strike);
return s;
}
```

4. Project Plan

Our project would not be finished without a detailed plan. The following plan was made at the beginning of our project to outline the main procedures of the project and specify where we were along the road.

4.1 Project process

4.1.1 Planning and Designing

The first phase of the project is coming up with the objective and basic design of the language, and making developing plans. The first draft of our LRM is the blue print of our language, we kept it up-to-date when we made modification to the language grammar. We meet on a weekly base to discuss the current status and next step of the project.

4.1.2 Development

We divided our developing phase into four major steps each along with their own testing process. The first step is to design AST, parser and scanner. Then the second step is to do type and scope checking by defining a SAST. The third step is the implementation of our code generator which generates the target code from our own language. The final step is using our language to develop some small domain specific applications that can demonstrate the advantages of our language.

4.1.3 Testing

Testing is done within every step along the progress to get feedback from every developing phase. Testing each phase individually and then test the whole integrated project ensured our project's validity while it is growing. Testing plans are discussed in detail in chapter 6.

4.2 Team Responsibilities

Fei Liu: Language guru
Taikun Liu: System architect
Jiayi Yan: Manager
Mengdi Zhang: Verification and validation

4.3 Project Timeline

The following time table recorded our main milestones. We regard these as deadlines for development.

Sep-10-2014	Team formed
Sep-20-2014	Language objective and proposal
Oct-13-2014	Scanner,Parser first draft
Oct-26-2014	LRM completed
Nov-06-2014	Parser, Scanner and AST completed
Nov-21-2014	SAST completed
Nov-27-2014	Code generation completed
Dec-03-2014	Testing Plan completed
Dec-12-2014	Final report finished

Software Development Environment

Our team developed out project on Mac OS X using OCaml 4.0 and OCamlyacc. And we used github for version control. We also wrote a shell script for running the test cases.

Project Logs and Personal Contributions

The following table lists actual progress in our development.

Sep-10-2014	Team formed
Sep-13-2014	Discussion of Language Objective
Sep-20-2014	Proposal Completed
Oct-04-2014	Parser and Lexer, first working version
Oct-11-2014	LRM first version
Oct-25-2014	LRM completed
Nov-01-2014	Parser and lexer updated
Nov-15-2014	SAST, TypeChecking, first working version
Nov-22-2014	Code generation, first version
Nov-27-Nov 30-2014	Testing, Modifying parser, typechecker, code-generater iteratively.
Dec-8-2014	Final report started
Dec-15-2014	Testing completed
Dec-17-2014	Final report completed

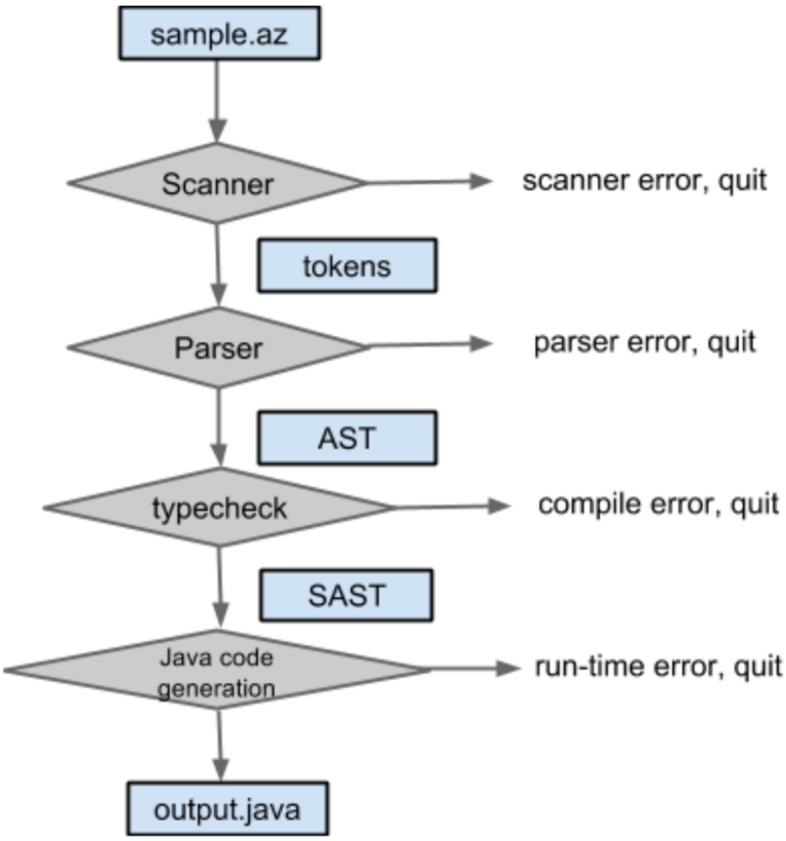
The following table lists personal contributions to this project

Scanner, Parser, AST	Jiayi Yan(major); Fei Liu; Taikun Liu
SAST, Typecheck	Taikun Liu
javagen, java codes	Fei Liu

Test cases, test scripts	Mengdi Zhang(major); Fei Liu; Taikun Liu
LRM, final report	Mengdi Zhang(major); Jiayi Yan(major); Taikun Liu; Fei Liu

5. Architectural Design

5.1 Architecture



As can be seen, the steps can be divided into the following sections:
1. Scanning
2. Parsing/AST
3. Typechecking/SAST
4. Java code generation

Scanning

The Scanner takes a file with extension of ".az" as input file and tokenizes the characters into meaningful symbols. Tokens include keywords, constant literals and operators used by angelaZ language. The process consist of discarding whitespace and comments and combining remaining character streams into tokens. Illegal character combination errors are caught here. The scanner is built with OCaml lexer.

Parsing/AST

The parser takes tokens provided by the scanner and generates an abstract syntax tree (AST). The process consists of defining operator precedence, pattern matching and pattern reduction. Syntax errors are caught here. The parser is built with OCaml yacc program.

An AST is an intermediate representation of the input file "sample.az" after parsing before semantic checking.

Type checking/SAST

Type checking takes AST provided by parsing and generates a type-safe semantic abstract syntax tree (SAST). The process consists of walking through the AST tree, checking and generating information such as type matching and scoping table. Semantic errors, such as type mismatching and using undeclared variables or functions are caught here.

For our project, since we have to check whether a field within a structure or option has been declared ahead. We use a different way to walk through the AST.

Our program consists of global statements and functions. So we first check global statements and function headers (without checking the statements inside functions at first). Secondly, our checking starts with main function. Checking all statements inside main function. If one statement calls another function, we then check whether that function exists, arguments match formals, arguments are well defined before go into that function to check that function's body statement. Consequently, functions that are not called will not be checked for their body statements.

An SAST is an intermediate type-safe representation of the input file "sample.az" after semantic checking.

Java code generation

Java code generation walks through SAST provided by type checking and generates a runnable java file "output.java". Errors found during evaluation of expressions, such as invalid index value -1, are caught here. Although this module generates java code, it does not compile it.

The output file "output.java" is a ready-to-run error-free Java file and can return the final result after being compiled by Java compiler.

5.2 The Runtime Environment

The runtime environment provides the infrastructure of operations in the generated java file. A set of runtime Java classes consist of implementation classes: "Structure. java", "Option.java", "MatrixMathematics.java" and "OptionMatrix.java"; exception handling classes: "NoSquareException.java" and "IllegalDimensionException.java"; print classes: "ToStirng.java".
The driver class is the Output.java. As dependencies, Matrix, Structure, and Option cover the types of the language and operations they have for only one such instance but not the interactions between them. For instance, Matrix multiplication is not covered because it is between Matrixes.. The MatrixMathematics.java is the file that invoke the java methods to do the actual computation. OptionMatrix.java is the file that contains the code for the Matrix-wise option pricing. The Exception suffix files are there for the Exceptions we may encounter. MatrixMathematics.java defines the operator methods that are between Matrixes.

The class declaration without the content of the classes is listed as follows:

```
public class Structure {
}

public class Option extends Structure {
}
public class MatrixMathematics {
}

public class OptionMatrix {
}

public class NoSquareException extends Exception {
}

public class IllegalDimensionException extends Exception {
}

public class ToString {
}
```

6. Testing Plan

6.1 Goals

Test cases are carefully selected to test all the development phases thoroughly without which the quality of the project is not ensured. The whole develope cycle is decomposed into four majors phases and we aimed at designing unit test on each modules and then run integration and system test on the whole integrated project. By running tests frequently, we could catch any backwards-incompatible changes during development. This system also allows test-driven development, as the test suite can be run with tests that use unimplemented language features, simply failing until those features are implemented.

6.2 Unit test

The aspects that we designed tests on during unit test are:
Variable declaration for each data type:
Int, Float, String, Boolean, Void, Matrix, Structure, Option
Legitimacy of identifiers

Variable assignment for each data type:
variable assigned by literals
variable assigned by variable
arithmetic assignment

Function declaration and calling:
main function
system function
parameter list

Operators:
arithmetic operators
boolean operators
structure access
option access
matrix access

Program structure:
 global variable declaration
 entrance function and user defined function

Statements:
 if/else statements
 for loops
 while loops

Type checking:
 for declarations
 for operators
 for statements
 matrix dimension matching

Scoping checking:
 uninitialized variable usage
 global and local variable usage

Others:
 comments
 type conversion
 precedence

6.3 Integration test

We designed test cases for integrated testing Parser/Scanner with type checking. Also we tested the correctness of semantic checking and code generation models together with the parser. We generated top level executable for semantic checking and code generation. The output of semantic checking is the SAST which is the SAT with types of all the variables resolved. The output of code generation is the java code.

7. Lesson Learned

When designing languages, syntax matters and one thing about syntax that shouldn't be considered is minimizing the number of keywords. The metric may cause much difficulty in distinguishing between language rules in parsing.
As a programmer, you are either writing code or fixing it. However, fixing code in OCaml can be painful, since debugging info is not informative when it comes to

shift/reduce error. Therefore, it's a wise choice to first design a language structure as comprehensive as possible, then begin with a small part of the functionality and add new features after making sure the the codes you've written are correct.

Even the best programmer cannot remember everything, so write it down. When discussing tokens, operators, logic and everything else about the language, write it down in good format. Order all rules neatly in Language Reference Manual before starting to code so that whenever you have doubt about anything rule, you can always check the LRM for reference.

Frequent and effective communication is a key to success of the project. Meet each week with your teammates to discuss problems you're having , the plan for the next week and so on. Also, meeting with TA each week to discuss the project is also helpful.

Some tools are recommended for the project, Github for version control and automated testing framework for debugging.

Last but not least, after finally coming up with a prototype next comes the hardest part, selling the language. A language no one uses is not a language, so prepare your presentation, documentation and tutorials well to show it off.

AngelaZ: Make it a world of Zen. Zen stands for peace and rest.

Appendix A

Test Cases

test-1.az

```
Int main (Int c, String s) {
}
```

test-2.az

```
Int i;
Boolean b;
Matrix m2(1,1);
Matrix m3(1,1);

Int main(Int argc, String argv) {
	Structure s = { a = "1", b = "2", c = "3" };
	 Matrix m(1,1);
	 m3 = (m +. m2) *.. 4;
	 Option o = {a = "1"};
	 o -> a;
	 argc = toInt(s -> a);
	 if (i == 2 || i > 0) {
		return argc;
	} else {
		return i+1;
	}
	while ( i >= 2 || (i < 100 && i != 3))
		argv = "argv";
	for (i = 0; i < 5; i = i + 1) {
		i = i * (5 + i);
	}
}

Void main2(Int argc2, Matrix m) {
	main(i, "str");
}
```

test-3.az

```
Int main(Int argc, String argv){
	Int a;
}
```

test-4.az

```
Boolean b1;
Boolean b2;
Boolean b3;
```

```
Boolean b4;
Int main(Int argc, String argv){
b1 = true;
b2 = false;
b3 = b1 && b2;
b4 = b1 || b2;
}
```

test-5.az

```
{* hello *}
Int a;
{**}
Int main(Int argc, String argv){
{*
hello
world
*}
}
```

test-6.az

```
Float f1;
Float f2;
Float f3;
Int main(Int argc, String argv){
f1 = 1;
f1 = 0.2;
f1 = 1e10;
f1 = 2E15;
f1 = 2e-9;
f1 = 2e+12;
f1 = 2E-0;
f1 = 2E+12;

f2 = f1 + 1;
f2 = f1 + 1.5;
f2 = 1.5 + f1;
f2 = f1 + 19e-10;
f2 = 1.5 + 19e10;

f2 = 9 - f1;
f2 = 10 - 9.3;
f2 = 9e10 - 8E+2;

f3 = f1 * f2;
f3 = f1 * 2;
```

```
f3 = f1 * 10e12;
f3 = 10e12 * f1;
f1 = 3e-19 * 1.5;

f1 = f2 / 2;
f1 = f2 /1.5;
f1 = 1.5 / f2;
f1 = f2 / 9e10;
f1 = 9e10 / 5E2;
f1 = 9e10 / 1.6;
}
```

test-7.az

```
Int main(Int argc, String argv){
Int a;
Boolean b1;
b1 = true;
for (a = 0; a < 10; a = a +1 ){
	Int b;
	b =1;
}
for (a = 0; a < 10; a = a +1 )
	Int c;

for (a = 0; b1; a = a +1 ){
	Int d;
	d =1;
}
}
```

test-8.az

```
Int main(Int argc, String argv){
Int asdf;
Int ASDF;
Int A0a9;
Int asdf____;
Int asdf_ASDF;
}
```

test-9.az

```
Int a;
Int b;
Int c;
Int main(Int argc, String argv){
b = 2;
```

```
a = 10;
if (a > b){
	c = 1;
}
else
	c = 0;

if (a > b)
	c = 1;
else
	c =0;

if (a > b){
	c =1;
}

if (a > b)
	c = 1;

if( a > b)
	c = 1;
else {
	c = 0;
}
if ( a >= b )
	c = 1;
if ( a < b)
	c = 1;
if ( a <= b)
	c = 1;
if ( a == b)
	c = 1;
if (a != b)
	c = 1;

Boolean b1;
Boolean b2;
b1 = true;
b2 = false;
if (b1)
	c = 1;
if (b1 && b2)
	c = 1;
if (b1 || b2)
```

```
    c = 1;
}
```

test-10.az

```
Int main(Int argc, String argv){
Int a;
a = 3;

Int b;
b = a + 5;
b = 5 + a;

Int c;
a = b - 1;
a = 1 - b;
c = a - b;

c = b;
c = b * 3;
c = a * b;

b = a / 2;
c = a / b;
c = 1 / a;
}
```

test-11.az

```
Matrix m1(2,2);
Matrix m2(2,2);
Matrix m3(2,2);
Int main(Int argc, String argv){
m1[1][1] = 2;
Float a;
a = m1[1][1];
m2 [1][1] = m1 [1][1];

m3 = m1 +. m2;
m2 = m2 -. m3;
m1 = m2 *. m3;
m3 = m1 /. m2;
}
```

test-12.az

```
Int main(Int argc, String argv){
Int a;
Int b;
Int c;
```

```
Boolean b1;
Boolean b2;
Boolean b3;

a = 2;
b = 1;

c = (a + b) * 3;
c = (a + b + 3) * 2;
c = 2* (a + b);
c = a + (1 + a);
c = (1 + a) * (1 + b);
c = (a + b);

b1 = true;
b2 = false;
b3 = b1 && (b1 || b2);
b3 = (b1 && b1 && b1);
}
```

test-13.az

```
Int main(Int argc, String argv){
String s;
s = " ";
s = "";
s = "asdf";
s = "!@asf#$%^as325df34&*()1sfd";
s = "as\\nasd";
s = "asd\r";
}
```

test-14.az

```
Int main(Int argc, String argv){
Int a;
Float b;
String c;
Matrix m(1,2);
Option o = {};
Structure s = {};
Boolean boo;
Void v;
}
```

test-15.az

```
Int main(Int argc, String argv){
Int a;
```

```
a = 5;
Boolean b1;
Boolean b2;
b1 = true;
b2 = false;

while (a > 0){
	a = a - 1;
}

while (b1 && b2)
	a = 10;
}
```

test-16.az

```
{* structure declaration *}
Int main(Int argc, String argv){
Structure s2 = { a="" };
Structure s3 = { a="a" };
String str;
str = "str";
Structure s4 = { a=str };
Structure s5 = { a=str, b = "b"};
}
```

test-17.az

```
{* structure access *}
Int main(Int argc, String argv){
String str;
str = "str";
Structure s = { a=str, b="b"};
s->a = s->b;
}
```

test-18.az

```
{* option declaration *}
Int main(Int argc, String argv){
Option o2 = { a="" };
Option o3 = { a="a" };
String str;
str = "str";
Option o4 = { a=str };
Option o5 = { a=str, b = "b"};
}
```

test-19.az

```
{* option access *}
```

```
Int main(Int argc, String argv){
String str;
str = "str";
Option o = { a=str, b="b"};
o->a = o->b;
}
```

fail-1.az

```
Int main(Int argc, String argv){
	Structure s = { a = "sdf"};
	foo(s);
}

Void foo (Structure s){
	print(s->b);
}
```

{*Fatal error: exception Failure("Field b does not exist within struct")*}

fail-2.az

```
Int main(Int argc, String argv){
Int a;
a = "asf";
}
```

{*Fatal error: exception Failure("variable a need to be assigned with same type int string")*}

fail-3.az

```
Int main(Int argc, String argv){
Int a;
a = 1.5;
}
```

{*Fatal error: exception Failure("variable a need to be assigned with same type int float")*}

fail-4.az

```
Int main(Int argc, String argv){
Int a;
a = asd;
}
```

{*Fatal error: exception Failure("Cannot find variable named asd")*}

fail-5.az

```
Int main(Int argc, String argv){
a = 2;
}
```

{*Fatal error: exception Failure("Cannot find variable named a")*}

fail-6.az

```
Int main(Int argc, String argv){
a = 1.2;
}
```

{*Fatal error: exception Failure("Cannot find variable named a")*}

fail-7.az

```
Int main(Int argc, String argv){
a = true;
}
```

{*Fatal error: exception Failure("Cannot find variable named a")*}

fail-8.az

```
Int main(Int argc, String argv){
Float a;
a = true;
}
```

{*Fatal error: exception Failure("variable a need to be float type")*}

fail-9.az

```
Int main(Int argc, String argv){
Float a;
a = asd;
}
```

{*Fatal error: exception Failure("Cannot find variable named asd")*}

fail-10.az

```
Int main(Int argc, String argv){
Float a;
a = "sasdf";
}
```

{*Fatal error: exception Failure("variable a need to be float type")*}

fail-11.az

```
Int main(Int argc, String argv){
Boolean b;
b = 0;
}
```

fail-12.az

```
Int main(Int argc, String argv){
Boolean b;
b = asd;
```

}

fail-13.az

```
Int main(Int argc, String argv){
String s;
s = 'asd';
}
```

fail-14.az

```
Int main(Int argc, String argv){
String s;
s = 12;
}
```

fail-15.az

```
String s;
s = True;
```

fail-16.az

```
Int a;
Float a;
```

fail-17.az

```
Int main(Int argc, String argv){
Int a;
Int b;
b = a+1.0;
}
```

fail-18.az

```
Int main(Int argc, String argv){
Float a;
a = 2;
Int f;
f = a;
}
```

fail-19.az

```
Int main(Int argc, String argv){
Int a;
a = 0;
Boolean b;
b = a;
}
```

fail-20.az

```
Matrix m;
```

fail-21.az

```
Int main(Int argc, String argv){
Matrix m(0,0);
```

```
}
```

fail-22.az

```
Int main(Int argc, String argv){
Matirx m (-1,2);
}
```

fail-23.az

```
Int main(Int argc, String argv){
Option o;
o = {a = "asdf"};
}
```

fail-24.az

```
Int main(Int argc, String argv){
Option o = { a = "asdf"; b = a;};
}
```

fail-25.az

```
Int main(Int argc, String argv){
Option o = {a = "sdf"};
o->a = 1;
}
```

fail-26.az

```
Int main(Int argc, String argv){
Option o {a=1};
}
```

fail-27.az

```
Int main(Int argc, String argv){
Option o { a ="asdf" };
o->b = "ad";
}
```

fail-28.az

```
Int main(Int argc, String argv){
Option o {a};
o->a = "asdf";
}
```

fail-29.az

```
Int main(Int argc, String argv){
String str = "asdf";
Option o {str};
}
```

fail-30.az

```
Int main(Int argc, String argv){
Option o = {a = "ad"};
Option o1;
o1 = o;
}
```

fail-31.az

```
Int main(Int argc, String argv){
String s;
s = "sdf";
String s1;
s1 = s;
String s2 = s1 + s;
}
```

fail-32.az

```
Int main(Int argc, String argv){
String s;
s = "sdf";
String s1;
s1 = s-"asdf";
}
```

fail-33.az

```
Int main(Int argc, String argv){
String s;
s = "sdf";
String s1;
s1 = s/1;
}
```

fail-34.az

```
Int main(Int argc, String argv){
String s;
s = "sdf";
String s1;
s1 = s*2;
}
```

AngelaZ: Make it a world of Zen. Zen stands for peace and rest.

AngelaZ: Make it a world of Zen. Zen stands for peace and rest.

AngelaZ: Make it a world of Zen. Zen stands for peace and rest.

Appendix B

Code:

```
{
  open Lexing
  open Parser
}
let digit = ['0'-'9']
let frac = '.' digit*
let exp = ['e' 'E'] ['-' '+']? digit+
let float = ['-']? digit* frac? exp?

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "{*"      { comment lexbuf } (* Comments *)
| '('       { LPAREN } (* precedence, matrix initialization, ifelseforwhile *)
| ')'       { RPAREN }
| '{'       { LBRACE } (* block of statements *)
| '}'       { RBRACE }
| '['       { LSQUARE } (* matrix index access *)
| ']'       { RSQUARE}
| ';'       { SEMI }
| ','       { COMMA }
| '+'       { PLUS }
| "+."      { MPLUS } (* plus matrix and matirx *)
| "+.."     { MIPLUS } (* plus matirx and int/float *)
| '-'       { MINUS }
| "-."      { MMINUS }
| "-.."     { MIMINUS }
| '*'       { TIMES }
| "*."      { MTIMES }
| "*.."     { MITIMES }
| '/'       { DIVIDE }
| "/."      { MDIVIDE }
| "/.."     { MIDIVIDE }
| '='       { ASSIGN }
| "&&"      { AND }
| "||"      { OR }
| "=="      { EQ }
| "!="      { NEQ }
| '<'       { LT }
| "<="      { LEQ }
| ">"       { GT }
| ">="      { GEQ }
| "->"      { ARROW } (* access struct/option element *)
| "if"      { IF }
| "else"    { ELSE }
| "for"     { FOR }
| "while"   { WHILE }
| "return"  { RETURN }
| "Boolean"{ BOOLEAN }
| "true"    { TRUE }
| "false"   { FALSE}
| "Matrix"  { MATRIX } (* variable type Matrix *)
| '''       { TRANSPOSE } (* option for Matrix *)
| '~'       { INVERSION }
| '^'       { DETERMINANT }
| "Structure" { STRUCTURE } (* variable type Structure *)
| "Option"  { OPTION } (* variable type Option *)
| "Int"     { INT }
| "Float"   { FLOAT }
| "String"  { STRING }
| "Void"    { VOID }
| ['-']?['0'-'9']+ as lxm { INT_LIT(int_of_string lxm) } (* integer literal *)
| float     { FLOAT_LIT (float_of_string (Lexing.lexeme lexbuf)) } (* float literal *)
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) } (* id/function name *)
| '"' (([' '-'!' '#'-'[' ']'-'~'] | '\\' ['\\' '"' 'n' 'r' 't'])* as s) '"'
                      { STRING_LIT(s) } (* string literal *)
| eof { EOF }
| _  { raise (Failure("illegal character " )) }

and comment = parse
  "*}" { token lexbuf }
| _    { comment lexbuf }
```

```
%{ open Ast %}

/* token */
%token SEMI COLON LPAREN RPAREN LSQUARE RSQUARE LBRACE RBRACE COMMA
%token PLUS MINUS MPLUS MMINUS MIPLUS MIMINUS TIMES DIVIDE MTIMES MDIVIDE MITIMES MIDIVIDE ASSIGN ARROW
%token EQ NEQ LT LEQ GT GEQ AND OR
%token RETURN IF ELSE FOR WHILE
%token BOOLEAN TRUE FALSE MATRIX STRUCTURE OPTION INT FLOAT STRING VOID
%token TRANSPOSE INVERSION DETERMINANT
%token <int> INT_LIT
%token <float> FLOAT_LIT
%token <string> STRING_LIT
%token <string> ID
%token EOF

/* associativity, precedence */
%nonassoc NOELSE
%nonassoc ELSE COMMA
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS MPLUS MMINUS MIPLUS MIMINUS
%left TIMES DIVIDE MTIMES MDIVIDE MITIMES MIDIVIDE
%left ARROW
%nonassoc TRANSPOSE INVERSION DETERMINANT

%start program
%type <Ast.program> program

%%

/* program consists of statements and function declarations */
program:
   { [], [] }
 | program stmt {($2 :: fst $1), snd $1 }
 | program fdecl { fst $1, ($2 :: snd $1) }

/* function declaration */
fdecl:
   retval formal_list RPAREN LBRACE stmt_list RBRACE
     { { func_name =  $1.vname;
         formals = List.rev $2;
         body = List.rev $5;
         ret =  $1.vtype
     } }

/* return structure */
retval:
    INT ID LPAREN  { { vtype=Int; vname=$2 } }
  | FLOAT ID LPAREN { { vtype=Float; vname= $2 } }
  | VOID ID LPAREN { { vtype=Void; vname=$2 } }
  | MATRIX ID LPAREN { { vtype=Matrix; vname=$2 } }
  | OPTION ID LPAREN { { vtype=Option; vname= $2 } }
  | STRUCTURE ID LPAREN { { vtype=Structure; vname= $2 } }
  | BOOLEAN ID LPAREN { { vtype=Boolean; vname=$2 } }
  | STRING ID LPAREN { { vtype=String; vname= $2 } }

formal_list:
    /* nothing */  { [] }
  | formal                  { [$1] }
  | formal_list COMMA formal { $3 :: $1 }

/* function arguments */
formal:
    tdecl      { $1 }
  | MATRIX ID  { { vname=$2; vtype=Matrix } }
  | OPTION ID  { { vname = $2; vtype = Option } }
  | STRUCTURE ID  { { vname = $2; vtype = Structure } }

/* primitive variable type declaration */
tdecl:
    INT ID { { vname = $2; vtype = Int } }
  | FLOAT ID  { { vname = $2; vtype = Float } }
  | VOID ID  { { vname = $2; vtype = Void } }
  | BOOLEAN ID  { { vname = $2; vtype = Boolean } }
  | STRING ID  { { vname = $2; vtype = String } }

/* matrix declaration */
mdecl:
    MATRIX ID LPAREN INT_LIT COMMA INT_LIT RPAREN { { mname = $2; mtype = Matrix; mrow = $4; mcol = $6 }  }

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

/* structure/option field declaration */
struct_arg:
```

```
  ID ASSIGN expr { { id = $1; value = $3 } }

struct_arg_list:
    /* nothing */  { [] }
  | struct_arg  { [$1] }
  | struct_arg_list COMMA struct_arg { $3 :: $1 }

/* statements */
stmt:
    expr SEMI { Expr($1) }
  | RETURN expr SEMI { Return($2) }
  | RETURN SEMI {Return(Noexpr)}
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
     { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
  | tdecl SEMI { Vardec($1) }
  | mdecl SEMI { Matdec($1) }
  | STRUCTURE ID ASSIGN LBRACE struct_arg_list RBRACE SEMI {Structdec($2, $5)}
  | OPTION ID ASSIGN LBRACE struct_arg_list RBRACE SEMI {Optiondec($2, $5)}

expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }

/* expressions */
expr:
    INT_LIT          { Int_lit($1) }
  | FLOAT_LIT         {Float_lit($1)}
  | STRING_LIT        {String_lit($1)}
  | TRUE         { Bool_lit(1) }
  | FALSE        { Bool_lit(0) }
  | ID               { Id($1) }
  | expr PLUS   expr { Binary_op($1, Add,   $3) }
  | expr MINUS  expr { Binary_op($1, Sub,   $3) }
  | expr TIMES  expr { Binary_op($1, Times,  $3) }
  | expr DIVIDE expr { Binary_op($1, Divide,   $3) }
  | expr MPLUS   expr { MatBinary_op($1, MAdd,   $3) }
  | expr MMINUS  expr { MatBinary_op($1, MSub,   $3) }
  | expr MTIMES  expr { MatBinary_op($1, MTime,  $3) }
  | expr MDIVIDE expr { MatBinary_op($1, MDivide,   $3) }
  | expr MIPLUS   expr { MatBinary_op($1, MIAdd,   $3) }
  | expr MIMINUS  expr { MatBinary_op($1, MISub,   $3) }
  | expr MITIMES  expr { MatBinary_op($1, MITime,  $3) }
  | expr MIDIVIDE expr { MatBinary_op($1, MIDivide,   $3) }
  | ID ASSIGN expr   { VarAssign($1, $3) }
  | ID LSQUARE expr RSQUARE LSQUARE  expr RSQUARE ASSIGN expr   { Matrix_element_assign($1, $3, $6, $9) }
  | ID ARROW ID ASSIGN expr { Struct_element_assign($1, $3, $5) }
  | ID LSQUARE expr RSQUARE LSQUARE  expr RSQUARE  { Matrix_element($1, $3, $6) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { Precedence_expr($2) }
  | ID ARROW ID {Struct_element($1, $3)}
/* bool_expr: */
  | expr EQ     expr { Binary_op($1, Eq, $3) }
  | expr NEQ    expr { Binary_op($1, Neq,   $3) }
  | expr LT     expr { Binary_op($1, Lt,  $3) }
  | expr LEQ    expr { Binary_op($1, Leq,   $3) }
  | expr GT     expr { Binary_op($1, Gt,  $3) }
  | expr GEQ    expr { Binary_op($1, Geq,   $3) }
  | expr AND    expr { Binary_op($1, And,   $3) }
  | expr OR     expr { Binary_op($1, Or,   $3) }
/* matrix_unary_option: */
  | expr TRANSPOSE   { MatUnary_op($1, MTranspose) }
  | expr INVERSION   { MatUnary_op($1, MInversion) }
  | expr DETERMINANT { MatUnary_op($1, MDeterminant) }

actuals_opt:
    /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

actuals_list:
    expr                    { [$1] }
  | actuals_list COMMA expr { $3 :: $1 }
```

```
type dataType =
  | Int
  | Float
  | String
  | Matrix
  | Option
  | Structure
  | Boolean
  | Void

type var_dec = {
  vname : string;
  vtype : dataType;
}

type mat_dec = {
  mname : string;
  mtype : dataType;
  mrow : int;
  mcol : int;
}

type bin_op = Add | Sub | Times | Divide| And | Or | Eq | Neq | Lt | Gt | Leq | Geq
type mat_op = MTime | MDivide |MAdd | MSub | MITime | MIDivide |MIAdd | MISub
type mat_uop = MTranspose | MInversion | MDeterminant

type expr =
    Binary_op of expr * bin_op * expr
  | MatBinary_op of expr * mat_op * expr
  | Id of string
  | Float_lit of float
  | Int_lit of int
  | String_lit of string
  | Call of string * expr list        (* function call: ID LPAREN expr COMMA expr RPAREN *)
  | VarAssign of string * expr        (* variable assign: ID ASSIGN expr *)
  | Matrix_element_assign of string * expr * expr * expr (* matrix element assign: ID LSQUARE expr RSQUARE LSQUARE expr RSQUARE ASSIGN expr *)
  | Struct_element_assign of string * string * expr      (* structu/option field assign ID ARROW ID ASSIGN expr *)
  | Matrix_element of string * expr * expr               (* matrix element access: ID LSQUARE expr RSQUARE LSQUARE expr RSQUARE *)
  | Precedence_expr of expr           (* precedenc: LPAREN expr RPAREN *)
  | Struct_element of string * string (* struct element access: ID ARROW ID *)
  | Bool_lit of int                   (* TRUE/FALSE stored as 1/0 *)
  | MatUnary_op of expr * mat_uop     (* matrix unary operation: expr TRANSPOSE/INVERSION/DETERMINANT *)
  | Noexpr

type struct_arg = {
  id : string;
  value : expr;
}

type stmt =
    Block of stmt list               (* block of stmts: LBRACE stmt_list RBRACE *)
  | Expr of expr                     (* fundamental stmt: expr SEMI *)
  | If of expr * stmt * stmt         (* if stmt: IF LPAREN expr RPAREN stmt *)
  | For of expr * expr * expr * stmt (* for stmt: FOR LPAREN expr SEMI expr SEMI expr RPAREN stmt *)
  | While of expr * stmt             (* while stmt: WHILE LPAREN expr RPAREN stmt *)
  | Return of expr                   (* return: RETURN expr SEMI *)
  | Vardec of var_dec                (* primitive varialbe declaration: INT/FLOAT/STRING/VOID/BOOLEAN ID *)
  | Matdec of mat_dec                (* matrix declaration: MATRIX ID LPAREN expr COMMA expr RPAREN *)
  | Structdec of string * struct_arg list (* structure declaration: STRUCTURE ID ASSIGN LBRACE arg RBRACE *)
  | Optiondec of string * struct_arg list (* option declaration: OPTION ID ASSIGN LBRACE arg RBRACE *)

type func_dec = {
  ret : dataType;
  func_name : string;
  formals : var_dec list;
  body : stmt list;
}

type program = stmt list * func_dec list



(* "Pretty printed" version of the AST, meant to generate a MicroC program
   from the AST.  These functions are only for pretty-printing (the -a flag)
   the AST and can be removed. *)

let string_of_dataType = function
    Int -> "int"
  | Float -> "float"
  | String -> "string"
  | Matrix -> "matrix"
  | Option -> "option"
  | Structure -> "structure"
  | Boolean -> "bool"
  | Void -> "void"

let string_of_vdecl vdecl =
  string_of_dataType vdecl.vtype ^ " " ^ vdecl.vname
let string_of_mdecl mdecl =
```

```
  string_of_dataType mdecl.mtype ^ " " ^ mdecl.mname ^
  "(" ^ string_of_int mdecl.mrow ^ ", " ^ string_of_int mdecl.mcol ^ ")"

let rec string_of_expr = function
    Float_lit(l) -> string_of_float l
  | Int_lit(l) -> string_of_int l
  | String_lit(l) -> "\"" ^ l ^ "\""
  | Id(s) -> s
  | Binary_op(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
      Add -> "+" | Sub -> "-" | Times -> "*" | Divide -> "/" | Eq -> "=="
      | Neq -> "!="       | Lt -> "<" | Leq -> "<=" | Gt -> ">" | Geq -> ">="
      | And -> "&&" | Or -> "||"
      ) ^ " " ^ string_of_expr e2
  | MatBinary_op(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
      MAdd -> "+." | MSub -> "-." | MTime -> "*." | MDivide -> "/."
      | MIAdd -> "+.." | MISub -> "-.." | MITime -> "*.." | MIDivide -> "/.."
      ) ^ " " ^ string_of_expr e2
  | VarAssign(v, e) -> v ^ " = " ^ string_of_expr e
  | Matrix_element_assign(id, r, c, e) -> id ^ "[" ^ string_of_expr r ^ " , " ^ string_of_expr c ^ "] = " ^ string_of_expr e
  | Struct_element_assign(id, i, e) -> id ^ " -> " ^ i ^ " = " ^ string_of_expr e
  | Matrix_element (v,e1,e2)->v ^ "[" ^ string_of_expr e1 ^ " , " ^ string_of_expr e2 ^ "] "
  | Call(f, el) -> f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> "void"
  | Precedence_expr(e) -> "( " ^ string_of_expr e ^ " )"
  | Struct_element(struct_id, element_id) -> struct_id ^ "->" ^ element_id
  | Bool_lit(b) -> (match b with | 1 -> " 1 " | 0 -> " 0 ")
  | MatUnary_op(e, o) ->
    (match o with
    MTranspose -> "Transpose" | MInversion -> "Inversion" | MDeterminant -> "Determinant"
    ) ^ "(" ^ string_of_expr e ^ ")"
  | _ -> "space"

let string_of_struct_arg arg = arg.id ^ " = " ^ string_of_expr arg.value

let rec string_of_stmt = function
    Block(stmts) -> "{\n  " ^ String.concat "  " (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n"
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n"
  | If(e, s1, s2) -> "if (" ^ string_of_expr e ^ ") " ^ string_of_stmt s1 ^ "else " ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
  | Vardec(vdecl) -> string_of_vdecl vdecl ^ ";\n"
  | Matdec(mdecl) -> string_of_mdecl mdecl ^ ";\n"
  | Structdec (id, argList) -> "Structure " ^ id ^ " = {" ^ String.concat ", " (List.map string_of_struct_arg (List.rev argList)) ^ "};\n"
  | Optiondec (id, argList) -> "Option " ^ id ^ " = {" ^ String.concat ", " (List.map string_of_struct_arg (List.rev argList)) ^ "};\n"

let string_of_fdecl fdecl =
  string_of_dataType fdecl.ret ^" " ^ fdecl.func_name ^ "(" ^
  String.concat ", " (List.map string_of_vdecl fdecl.formals) ^ ")\n{\n" ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^ "}\n"

let string_of_program (stmts, funcs) =
  String.concat "\n" (List.map string_of_stmt (List.rev stmts)) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl (List.rev funcs)) ^ "\n"
```

```
open Ast
open Sast
(*************************************Collection symbol table*)

type struc_table = {
  struc_name : string; (*name of a structure*)
  mutable struc_fields : string list; (*keys within a structure*)
}
type option_table = {
  option_name : string; (*name of a option*)
  mutable option_fields : string list; (*keys within a option*)
}
type size_of_matrix = {
    rows : int; (*number of rows*)
    cols : int; (*number of cols*)
}
type matrix_table = {
  matrix_name : string; (*name of a matrix*)
  msize : size_of_matrix; (*size of a matrix*)
}
(***************************************** Symbol Tables *)

type symbol_table = { (*general symbol table for variables*)
  parent : symbol_table option;
  mutable variables : (string * Ast.dataType) list;
  mutable structs : struc_table list;
  mutable options : option_table list;
  mutable matrixes : matrix_table list;
}

type environment = {
  mutable func_return_type : Ast.dataType; (* Function return type *)
  scope : symbol_table;        (* symbol table for varibles *)
  mutable functions : (string * Ast.dataType list * Ast.dataType) list; (* symbol table for global functions, nested function declaration not supported*)
}
(******************************************* Debug Functions*)
let print_var elem = print_endline (fst elem ^ "\t" ^ string_of_dataType (snd elem)^" is in scope")

let rec print_vars_list = function (*string*dataType list to print*)
[] -> print_string "empty vars\n"
| e::l -> print_var e ; print_vars_list l

let rec print_mvars = function
  [] -> print_endline "empty matrices"
  |e::l -> print_endline e.matrix_name ; print_mvars l
(******************************************* Initial Functions*)

let new_env : environment =
  let core =[("toString", [String], String);("toString", [Int], String);
            ("toString", [Float], String);("toString", [Matrix], String);
            ("toString", [Structure], String);("toString", [Option], String);
            ("toString", [Boolean], String);("printM", [Matrix], Void);
            ("print", [String], Void);("print", [Int], String);
            ("print", [Float], Void);("print", [Matrix], Void);
            ("print", [Structure], Void);("print", [Option], Void);
            ("print", [Boolean], Void);("toInt", [String], Int);
            ("toFloat", [String], Float);("price", [Option], Float);
            ("toBoolean", [String], Boolean);
            ("priceM", [Matrix;Matrix;Matrix;Matrix;Matrix], Matrix);] in (*built in functions*)
  let s = { variables = [];structs = []; options = []; matrixes = []; parent = None }
  in
  {scope = s ; func_return_type = Void; functions = core;}

let inner_scope (env : environment) : environment =
  let s = { variables = []; structs = []; options = [];matrixes = []; parent = Some(env.scope) } in
  { env with scope = s; }

(******************************************* Utils*)

let rec check_dup l = match l with
        [] -> false
      | (h::t) ->
        let x = (List.filter (fun x -> x = h) t) in
        if (x == []) then
           check_dup t
        else
           true

let rec samelists l1 l2 = match l1, l2 with
| [], []       -> true
| [], _ | _, []        -> false
| x::xs, y::ys -> x = y && samelists xs ys

let is_keyword (var_name:string) =
  let keyword_set = ["main";"Void";"if";"else";"for";"while";"return";"true";"false";
  "Int";"Float";"Matrix";"Structure";"Boolean";] in
  try
    List.find (fun x -> x=var_name ) keyword_set;
```

```
    true
  with Not_found -> false

let rec find_vars_exist (env_scope : symbol_table) (var_name : string) =
  try
    List.find (fun (s, _) -> s = var_name) env_scope.variables;
    true
  with Not_found ->
    try
      List.find (fun x ->x.struc_name =var_name) env_scope.structs;
      true
    with Not_found ->
      try
        List.find (fun x->x.option_name = var_name) env_scope.options;
        true
      with Not_found ->
        try
          List.find (fun x->x.matrix_name = var_name) env_scope.matrixes;
          true
        with Not_found ->
          (match env_scope.parent with
          | Some(p) -> find_vars_exist p var_name
          | _ -> false)

let rec find_vars (env_scope : symbol_table) (var_name : string) : Ast.dataType=
  try
    let (_, typ) = List.find (fun (s, _) -> s = var_name) env_scope.variables in
    typ
  with Not_found ->
    try
      List.find (fun x ->x.struc_name =var_name) env_scope.structs;
      Structure
    with Not_found ->
      try
        List.find (fun x->x.option_name = var_name) env_scope.options;
        Option
      with Not_found ->
        try
          List.find (fun x->x.matrix_name = var_name) env_scope.matrixes;
          Matrix
        with Not_found ->
          (match env_scope.parent with
          | Some(p) -> find_vars p var_name
          | _ -> raise(Failure("Cannot find variable named " ^ var_name) ))

(*The below three functions return cooresponding dataType table instead of dataType or Boolean*)
let rec find_structs (env_scope : symbol_table) (var_name : string) =
  try
    let struct_find = List.find (fun x -> x.struc_name = var_name) env_scope.structs in
    struct_find
  with Not_found ->
    match env_scope.parent with
        | Some(p) -> find_structs p var_name
        | _ -> raise(Failure("Cannot find strucure named " ^ var_name) )

let rec find_options (env_scope : symbol_table) (var_name : string) =
  try
    let option_find = List.find (fun x -> x.option_name = var_name) env_scope.options in
    option_find
  with Not_found ->
    match env_scope.parent with
        | Some(p) -> find_options p var_name
        | _ -> raise(Failure("Cannot find option named " ^ var_name) )

let rec find_matrix (env_scope : symbol_table) (var_name : string) =
  try
    let matrix_find = List.find (fun x -> x.matrix_name = var_name) env_scope.matrixes in
    matrix_find
  with Not_found ->
    match env_scope.parent with
        | Some(p) -> find_matrix p var_name
        | _ -> raise(Failure("Cannot find matrix named " ^ var_name) )

let find_funcs_exist (env : environment) (var_name : string) =
  try
    List.find (fun (s, _, _) -> s = var_name) env.functions;
    true
  with Not_found -> false

let find_funcs (env : environment) (var_name : string) (arglist : Ast.dataType list)=
  try
    let (_, _, typ) = List.find (fun (s, forlist, _) ->
                                s = var_name && samelists forlist arglist) env.functions in
    typ
  with Not_found -> raise(Failure("Cannot find function named " ^ var_name) )

(*get type of an expression*)
let get_type (e : Sast.expr_t): Ast.dataType =
  match e with
```

```
  Binary_op_t(_,_,_,t) -> t
 | MatBinary_op_t(_,_,_,t) -> t
 | Id_t(_,t) -> t
 | Float_lit_t(_,t) -> t
 | Int_lit_t(_,t) -> t
 | String_lit_t(_,t) -> t
 | Call_t(_,_, t) ->t
 | VarAssign_t(_,_,t)->t
 | Matrix_element_assign_t (_,_,_,_,t) -> t
 | Struct_element_assign_t (_,_,_,t) -> t
 | Matrix_element_t(_,_,_,t)->t
 | Precedence_expr_t(_,t)->t
 | Struct_element_t(_,_,t)->t
 | Bool_lit_t(_,t)->t
 | MatUnary_op_t(_,_,t) ->t
 | Noexpr_t(t)->t

(*get dimension of matrix *)
let rec get_dimension (env: environment) (exp : Ast.expr) : size_of_matrix =
 match exp with
 | Id(s) ->
   let m = find_matrix env.scope s in
   m.msize
 | MatBinary_op (e1, op, e2) -> get_dimension env e1
 | VarAssign (s,e) -> let exp = Id(s) in get_dimension env exp
 | Precedence_expr (e) -> get_dimension env e
 | MatUnary_op (e,op) ->
   let msize = get_dimension env e in
   (match op with
   | MTranspose -> {rows = msize.cols; cols = msize.rows}
   | MInversion -> msize
   | MDeterminant -> raise(Failure("Not of matrix type")))
 | _ -> raise(Failure("Cannot find dimension which is not of matrix type"))

(*test matrix sizes are equal*)
let size_equal (size1 : size_of_matrix) (size2 : size_of_matrix) : bool =
 if (size1.rows = size2.rows) && (size1.cols = size2.cols)
   then true
 else false

(*test matrix sizes are transpose equal*)
let size_mult_equal (size1 : size_of_matrix) (size2 : size_of_matrix) : bool =
 if (size1.rows = size2.cols) && (size1.cols = size2.rows)
   then true
 else false

let get_formal_type (vardec : Ast.var_dec) : Ast.dataType  = vardec.vtype

(****************************************** Annotate and Check*)
let rec annotate_expr (env : environment) (e : Ast.expr) (func_ts : Sast.func_dec_t list): Sast.expr_t =
 match e with
 Binary_op(e1, op, e2) ->
   let e1_a = annotate_expr env e1 func_ts in
   let e2_a = annotate_expr env e2 func_ts
   in
     let e1_t = get_type e1_a in
     let e2_t = get_type e2_a in
     (match op with
     | Add ->
       (match e1_t with
       | Float ->
         if (e2_t = Float) || (e2_t = Int)
         then Binary_op_t(e1_a, op, e2_a, Float)
         else raise(Failure("Binary operation has un-consistent types"))
       | Int ->
         (match e2_t with
         | Int -> Binary_op_t(e1_a, op, e2_a, Int)
         | Float -> Binary_op_t(e1_a, op, e2_a, Float)
         | _ -> raise(Failure("Binary operation has un-consistent types")))
       | String ->
         if  e2_t = String
         then Binary_op_t(e1_a, op, e2_a, e1_t)
         else raise(Failure("Binary operation has un-support types")))
     | Sub | Times | Divide ->
       (match e1_t with
       | Float ->
         if (e2_t = Float) || (e2_t = Int)
         then Binary_op_t(e1_a, op, e2_a, Float)
         else raise(Failure("Binary operation has un-consistent types"))
       | Int ->
         (match e2_t with
         | Int -> Binary_op_t(e1_a, op, e2_a, Int)
         | Float -> Binary_op_t(e1_a, op, e2_a, Float)
         | _ -> raise(Failure("Binary operation has un-consistent types")))
       | _ ->  raise(Failure("Binary operation has unvalid types")))
     | And | Or   ->
       if (e1_t<>Boolean) || (e2_t<>Boolean)
         then raise(Failure("Boolean should be the types around boolean operations"))
       else Binary_op_t(e1_a, op, e2_a, e1_t)
```

```
    | Eq | Neq | Lt | Gt | Leq | Geq ->
      if (e1_t<>e2_t)
        then raise(Failure("Comparasion can only happen between same type."))
      else Binary_op_t(e1_a, op, e2_a, Boolean)
      )
| MatBinary_op(e1, op, e2) ->
  let e1_a = annotate_expr env e1 func_ts in
  let e2_a = annotate_expr env e2 func_ts in
    let e1_t = get_type e1_a in
    let e2_t = get_type e2_a in
    (match op with
    | MTime | MDivide |MAdd | MSub ->
      if (e1_t <> Matrix) || (e2_t <> Matrix)
      then raise(Failure("Matrix operation has to be Matrix type on both sides"))
      else
        let sizel = get_dimension env e1 in
        let sizer = get_dimension env e2 in
        (match op with
        | MAdd | MSub ->
          if size_equal sizel sizer
            then MatBinary_op_t(e1_a, op, e2_a, Matrix)
          else raise(Failure("Matrix addition or Matrix subtraction has incompitable sizes"))
        | MTime | MDivide ->
          if size_mult_equal sizel sizer
            then MatBinary_op_t(e1_a, op, e2_a, Matrix)
          else raise(Failure("Matrix addition or Matrix subtraction has incompitable sizes"))
          )
    | MITime | MIDivide |MIAdd | MISub ->
      if (e1_t <> Matrix) || ((e2_t <> Int) && (e2_t <> Float))
      then raise(Failure("MatrixElment operation has to be Matrix type on left side and Integer on the right side"))
      else MatBinary_op_t(e1_a, op, e2_a, Matrix)
      )
| Id(s) ->
  let typ = find_vars env.scope s in Id_t(s,typ)
| Float_lit(f) ->Float_lit_t(f,Float)
| Int_lit(i) ->Int_lit_t(i,Int)
| String_lit(s) ->String_lit_t(s,String)
| Call(name, exl)-> (
    let exl_a = List.map (fun x -> annotate_expr env x func_ts) exl in
    let arglist = List.map (fun x -> get_type x) exl_a in
    let ret_type = find_funcs env name arglist in (*Check whether this func exist*)
    try
      let func_dec_t = List.find (fun x -> x.func_name_t = name) func_ts in
      (*Check for arguments and added to environment*)
      let formals = func_dec_t.formals_t in
      let arg_size  = List.length exl in
      let inner_env = inner_scope env in
      inner_env.func_return_type <- ret_type;
      for i = 0 to arg_size-1 do
        let formal = List.nth formals i in
        let arg = List.nth exl i in
        (match formal.vtype with
        | Int | Float | Boolean | String ->
          inner_env.scope.variables <- [(formal.vname, formal.vtype)]@inner_env.scope.variables
        | Matrix ->
          (match arg with
          | Id(sn) ->
            let mat_to_func = find_matrix env.scope sn in
            inner_env.scope.matrixes <- {mat_to_func with matrix_name = sn} :: inner_env.scope.matrixes
          | _ -> raise(Failure("Cannot pass this variable")))
        | Structure ->
          (match arg with
          | Id(sn) ->
            let struct_to_func = find_structs env.scope sn in
            inner_env.scope.structs <- {struct_to_func with struc_name = sn} :: inner_env.scope.structs
          | _ -> raise(Failure("Cannot pass this variable")))
        | Option ->
          (match arg with
          | Id(sn) ->
            let option_to_func = find_options env.scope sn in
            inner_env.scope.options <- {option_to_func with option_name = sn} :: inner_env.scope.options
          | _ -> raise(Failure("Cannot pass this variable")))
        )
      done;
      let stmtlist = func_dec_t.ori_body in
      func_dec_t.body_t = annotate_stmts inner_env stmtlist func_ts;
      Call_t(name,exl_a,ret_type)
  with Not_found ->Call_t(name,exl_a,ret_type) (*Must be built-in function*)
)
| VarAssign(s, e2)->  let exp = Id(s) in annotate_assign env exp e2 func_ts
| Matrix_element_assign(s, e1, e2, e3) ->
  let exp = Matrix_element(s, e1, e2) in annotate_assign env exp e3 func_ts
| Struct_element_assign(s1, s2, e) ->
  let exp = Struct_element(s1, s2) in annotate_assign env exp e func_ts
| Precedence_expr(e) -> annotate_expr env e func_ts
| Struct_element(s1,s2) -> check_struc_elem env s1 s2
| Matrix_element(s,me1,me2) -> check_matrix_elem env s me1 me2 func_ts
| Bool_lit(e1) -> Bool_lit_t(e1,Boolean)
| MatUnary_op(e,op) ->
```

```
    let e_a = annotate_expr env e func_ts in
    let e_t = get_type e_a in
    if e_t <> Matrix
     then raise(Failure("Matrix uni-operation has to be applied to Matrix type"))
    else
     (match op with
   | MTranspose | MInversion -> MatUnary_op_t(e,op,Matrix)
   | MDeterminant -> MatUnary_op_t(e,op,Float))
 | _ -> Noexpr_t(Void)

and check_matrix_elem (env : environment) (name:string)(e1:Ast.expr)(e2:Ast.expr)(func_ts : Sast.func_dec_t list) : Sast.expr_t =
  let var_type = find_vars env.scope name in
  match var_type with
  | Matrix ->
    let e1_a = annotate_expr env e1 func_ts in
    let e2_a = annotate_expr env e2 func_ts in
    let e1_typ = get_type e1_a in
    let e2_typ = get_type e2_a in
     if (e1_typ<>Int)||(e2_typ<>Int)
      then raise(Failure("Indexes of Matrix must be of Int type"))
     else Matrix_element_t(name,e1_a,e2_a,Float)
  | _ -> raise(Failure("Should be of Matrix type for accessing elements"))

and check_struc_elem (env:environment) (name:string)(key:string) : Sast.expr_t =
  let var_type = find_vars env.scope name in
  match var_type with
  | Structure ->
    (let struc = find_structs env.scope name in
    try
      List.find (fun s -> s = key) struc.struc_fields;
      Struct_element_t(name,key,String)
    with Not_found -> raise(Failure("Field "^key^" does not exist within struct")))
  | Option ->
    (let opt = find_options env.scope name in
    try
      List.find (fun s -> s = key) opt.option_fields;
      Struct_element_t(name,key,String)
    with Not_found -> raise(Failure("Field "^key^" does not exist within struct")))
  | _ -> raise(Failure("Should be of Structure type for accessing fields"))

and annotate_assign (env : environment) (e1 : Ast.expr) (e2 : Ast.expr)(func_ts : Sast.func_dec_t list) : Sast.expr_t =
  let e2_a = annotate_expr env e2 func_ts in
  match e1 with
  | Id(x) ->
    let e2_type = get_type e2_a in
    let e1_type = find_vars env.scope x in
    (match e1_type with
      | Float ->
        if (e2_type = Float) || (e2_type = Int)
        then VarAssign_t(Id_t(x, e1_type), e2_a, e1_type)
        else raise(Failure("variable "^x^" need to be float type"))
      | _ ->
        if e2_type <> e1_type
        then raise(Failure("variable "^x^" need to be assigned with same type " ^ string_of_dataType e1_type ^ " " ^ string_of_dataType e2_type ))
        else VarAssign_t(Id_t(x, e1_type), e2_a, e2_type)
      )
  | Matrix_element(s,me1,me2)->
    let elem_t = check_matrix_elem env s me1 me2 func_ts in
    let e2_type = get_type e2_a in
    (match e2_type with
    |Float|Int ->
      let me1_a = annotate_expr env me1 func_ts in
      let me2_a = annotate_expr env me2 func_ts in
      if (get_type me1_a = Int) && (get_type me2_a = Int)
        then Matrix_element_assign_t(s, me1_a, me2_a, e2_a, Float)
      else raise(Failure("Matrix indexing has to be of type int"))
    | _ -> raise(Failure("Only allow Float or Int assigned to Matrix element"))
    )
  | Struct_element(s1,s2)->
    check_struc_elem env s1 s2;
    let value_type = get_type e2_a in
    (match value_type with
    |String -> Struct_element_assign_t(s1,s2,e2_a,String)
    |_ ->raise(Failure("Only String type can be assigned to structure"))
    )
  | _ -> raise(Failure("Assignment need to be applied to a variable"))

and annotate_stmt (env : environment) (s : Ast.stmt) (func_ts : Sast.func_dec_t list): Sast.stmt_t =
  match s with
    Block(stmtlist) ->
      let env_inner = inner_scope env
      in
        let stmt_t_list =  annotate_stmts env_inner stmtlist func_ts
        in
          Block_t(stmt_t_list)
    | Expr(e) ->
      let expr_a = annotate_expr env e func_ts
      in
        Expr_t(expr_a)
```

```
  | If(be, body1, body2) ->
    let be_a = annotate_expr env be func_ts in
      if (get_type be_a) <> Boolean
        then raise(Failure("there should be boolean expression within If"))
      else
      let body1_a = annotate_stmt env body1 func_ts in
      let body2_a = annotate_stmt env body2 func_ts in
        If_t(be_a, body1_a, body2_a)
  | For(ae1, be, ae2, body)->
    let ae1_a =
      (match ae1 with
      | VarAssign(s, e2) -> let exp = Id(s) in annotate_assign env exp e2 func_ts
      | Matrix_element_assign(s, e1, e2, e3) ->
        let exp = Matrix_element(s, e1, e2) in annotate_assign env exp e3 func_ts
      | Struct_element_assign(s1, s2, e) ->
        let exp = Struct_element(s1, s2) in annotate_assign env exp e func_ts
      | Noexpr -> Noexpr_t(Void)
      | _ -> raise(Failure("Need assignment in for loop header"))) in
    let be_a =
      (match be with
      | Binary_op(_,_,_) -> annotate_expr env be func_ts
      | Id(s) ->
        let ret_type = find_vars env.scope s in
        if ret_type <> Boolean
          then raise(Failure("condition expression within For loop is not a correct type"))
        else
          annotate_expr env be func_ts
      | Bool_lit(_) -> annotate_expr env be func_ts
      | Noexpr -> Noexpr_t(Void)
      | _ -> raise((Failure("condition expression within For loop is not a correct type")))) in
    let ae2_a =
      (match ae2 with
      | VarAssign(s, e2) -> let exp = Id(s) in annotate_assign env exp e2 func_ts
      | Matrix_element_assign(s, e1, e2, e3) ->
        let exp = Matrix_element(s, e1, e2) in annotate_assign env exp e3 func_ts
      | Struct_element_assign(s1, s2, e) ->
        let exp = Struct_element(s1, s2) in annotate_assign env exp e func_ts
      | Noexpr -> Noexpr_t(Void)
      | _ -> raise((Failure("there should be assign expression within for loop")))) in
    let body_a = annotate_stmt env body func_ts
      in
        if (get_type be_a<>Boolean) && (get_type be_a <> Void)
          then raise(Failure("there should be boolean expression within For"))
        else
        For_t(ae1_a, be_a, ae2_a, body_a)
  | While(be, body)  ->
    let be_a = annotate_expr env be func_ts in
    let body_a = annotate_stmt env body func_ts
      in
        if (get_type be_a<>Boolean) && (get_type be_a <> Void)
          then raise(Failure("there should be boolean expression within For"))
        else
        While_t(be_a, body_a)
  | Return(e) ->
    let e_a = annotate_expr env e func_ts
    in
      let return_type = get_type e_a
      in
        if return_type <> env.func_return_type
        then raise(Failure("actual return type is not same with function return type: " ^ string_of_dataType env.func_return_type))
        else
          Return_t(e_a)
  | Vardec(vd) ->
    let var_type = vd.vtype in
    let var_name = vd.vname
    in
      if is_keyword var_name
       then raise(Failure("Cannot use keyword " ^ var_name ^ " as variable name"))
      else
        let exist_v = find_vars_exist env.scope var_name
        in
          if exist_v
            then raise(Failure("Variable name " ^ var_name ^ " already been used."))
          else
            (
              env.scope.variables <- env.scope.variables@[(var_name, var_type)];
            Vardec_t(vd, var_type))
  | Matdec(md) ->
    let var_type = md.mtype in
    let var_name = md.mname
    in
      if is_keyword var_name
        then raise(Failure("Cannot use keyword " ^ var_name ^ "  as variable name"))
      else
        if var_type <> Matrix
         then raise(Failure("The declaration of  " ^ var_name ^ " only apply to Matrix"))
        else
          let exist_v = find_vars_exist env.scope var_name
          in
```

```
            if exist_v
              then raise(Failure("Variable name " ^ var_name ^ "  already been used."))
            else
              if (md.mrow < 1) ||  (md.mcol < 1)
                then raise(Failure("Invalid number of rows or cols for Mat declare."))
              else
              ( env.scope.matrixes <-{matrix_name = var_name; msize = {rows = md.mrow; cols = md.mcol}} :: env.scope.matrixes;
              Matdec_t(md, var_type))
  | Structdec(var_name,starglist) ->
    if is_keyword var_name
        then raise(Failure("Cannot use keyword " ^ var_name ^ "  as variable name"))
    else
        let exist_v = find_vars_exist env.scope var_name
        in
          if exist_v
              then raise(Failure("Variable name " ^ var_name ^ "  already been used."))
          else
            (*Check whether filed names have duplicates and type are of string*)
            let fields = List.map (fun x ->
                          let fid = x.id in
                          let fval = annotate_expr env x.value func_ts in
                          if get_type fval <> String
                            then raise(Failure("Only String can be assigned to filed of Structure"))
                          else fid ) starglist
            in
            let dup = check_dup fields
            in
              if dup
                then raise(Failure("There are duplicate fields inside structure "^var_name))
              else
                ( env.scope.structs<-env.scope.structs@[{struc_name = var_name; struc_fields = fields}];
              Structdec_t(var_name, starglist, Structure))
  | Optiondec(var_name,starglist) ->
    if is_keyword var_name
        then raise(Failure("Cannot use keyword " ^ var_name ^ "  as variable name"))
    else
        let exist_v = find_vars_exist env.scope var_name
        in
          if exist_v
              then raise(Failure("Variable name " ^ var_name ^ "  already been used."))
          else
            (*Check whether filed names have duplicates and type are of string*)
            let fields = List.map (fun x ->
                          let fid = x.id in
                          let fval = annotate_expr env x.value func_ts in
                          if get_type fval <> String
                            then raise(Failure("Only String can be assigned to filed of Structure"))
                          else fid ) starglist
            in
            let dup = check_dup fields
            in
              if dup
                then raise(Failure("There are duplicate fields inside Option "^var_name))
              else
                ( env.scope.options<-env.scope.options@[{option_name=var_name; option_fields=fields}];
              Optiondec_t(var_name, starglist, Option))

and annotate_stmts  (env : environment) (stmts : Ast.stmt list) (func_ts : Sast.func_dec_t list): Sast.stmt_t list =
  List.map (fun x -> (print_string (string_of_stmt x);annotate_stmt env x func_ts)) stmts

let annotate_global_stmts (env : environment) (stmts : Ast.stmt list) : Sast.stmt_t list =
  List.map (fun x ->
    (match x with
    |Vardec(_)| Matdec(_)| Structdec(_,_)| Optiondec (_,_) -> annotate_stmt env x []
    | _ -> raise(Failure("Global statements can only be variable declaration")))
    ) stmts

let insert_func (env: environment) (func : Ast.func_dec) =
  let ret_type = func.ret in
  let name = func.func_name in
    if is_keyword name && name <> "main"
        then raise(Failure("Cannot use keyword " ^ name ^ " as function name"))
    else
        let exist_f = find_funcs_exist env name
          in
            if exist_f
              then raise(Failure("Function name " ^ name ^ " already been used."))
            else
              let formals_list = func.formals in
              let data_type_list = List.map (fun x-> get_formal_type x) formals_list in
              env.functions <- env.functions @ [(name,data_type_list,ret_type)]

let annotate_func (env: environment) (func : Ast.func_dec) : Sast.func_dec_t =
    {ret_t = func.ret; func_name_t = func.func_name; formals_t = func.formals; body_t = []; ori_body = func.body}

let annotate_funcs  (env : environment) (funcs : Ast.func_dec list)  : Sast.func_dec_t list=
  List.iter (fun x -> insert_func env x) funcs;
  List.map (fun x -> annotate_func env x) funcs
```

```
let check_from_main (env : environment) (func_ts : Sast.func_dec_t list) =
  try
    let mainfunc = List.find (fun x -> x.func_name_t = "main") func_ts in
    let formals = mainfunc.formals_t in
    let size = List.length formals in
    for i = 0 to size-1 do
          let formal = List.nth formals i in
          print_endline (string_of_dataType formal.vtype);
          env.scope.variables <- [(formal.vname, formal.vtype)]@env.scope.variables;
          env.func_return_type <- Int
    done;
    let mainbody = mainfunc.ori_body in
    mainfunc.body_t <- annotate_stmts env mainbody func_ts;
    print_endline "\nSemantic analysis completed successfully."
  with Not_found -> ()

(******************************************** Checking Program and main function exists *)
let check_program (stmts, funcs) =
  let globals = List.rev stmts in
  let functions = List.rev funcs in
  let env = new_env in
    annotate_global_stmts env globals;
    let func_dec_t_list = annotate_funcs env functions in
    let main_exist = find_funcs_exist env "main" in
    if main_exist
      then check_from_main env func_dec_t_list
    else
      raise(Failure("Must declare main function"))
```

```
open Ast
(*************************
**** SAST ****************
*************************)

(*****Types Annotated******)
type expr_t =
   Binary_op_t of expr_t * bin_op * expr_t * dataType
  | MatBinary_op_t of expr_t * mat_op * expr_t * dataType
  | Id_t of string * dataType
  | Float_lit_t of float * dataType
  | Int_lit_t of int * dataType
  | String_lit_t of string * dataType
  | Call_t of string * expr_t list * dataType
  | VarAssign_t of expr_t * expr_t * dataType
  | Matrix_element_assign_t of string * expr_t * expr_t * expr_t * dataType
  | Struct_element_assign_t of string * string * expr_t * dataType
  | Matrix_element_t of string * expr_t * expr_t * dataType
  | Precedence_expr_t of expr_t * dataType
  | Struct_element_t of string * string * dataType
  | Bool_lit_t of int * dataType
  | MatUnary_op_t of expr * mat_uop * dataType
  | Noexpr_t of dataType

type stmt_t =
    Block_t of stmt_t list
  | Expr_t of expr_t
  | If_t of expr_t * stmt_t * stmt_t
  | For_t of expr_t * expr_t * expr_t * stmt_t
  | While_t of expr_t * stmt_t
  | Return_t of expr_t
  | Vardec_t of var_dec * dataType
  | Matdec_t of mat_dec * dataType
  | Structdec_t of string * struct_arg list * dataType
  | Optiondec_t of string * struct_arg list * dataType

type func_dec_t = {
  ret_t : dataType;
  func_name_t : string;
  formals_t : var_dec list;
  mutable body_t : stmt_t list;
  mutable ori_body : stmt list;
}

type program_t = stmt_t list * func_dec_t list
```

```
open Printf
open Ast

(* data type string gen*)
let string_of_dataType = function
    Int -> "int"
  | Float -> "double"
  | String -> "String"
  | Matrix -> "Matrix"
  | Option -> "Option"
  | Structure -> "Structure"
  | Boolean -> "boolean"
  | Void -> "void"

(* varaible declaration string gen*)
let string_of_vdecl vdecl =  string_of_dataType vdecl.vtype ^ " " ^ vdecl.vname ^ match vdecl.vtype with
| Int -> "=0"
  | Float-> "=0.0"
  | String -> "=null"
  | Matrix-> ""
  | Option -> ""
  | Structure -> ""
  | Boolean -> "=false"
  | Void  -> ""
(* variable declaration string gen notice the static *)
let string_of_global_vdecl vdecl =  "static "^string_of_dataType vdecl.vtype ^ " " ^ vdecl.vname ^ match vdecl.vtype with
| Int -> "=0"
  | Float-> "=0.0"
  | String -> "=null"
  | Matrix-> ""
  | Option -> ""
  | Structure -> ""
  | Boolean -> "=false"
  | Void  -> ""


let string_of_vdecl_argument vdecl =  string_of_dataType vdecl.vtype ^ " " ^ vdecl.vname
let string_of_mdecl mdecl = string_of_dataType mdecl.mtype ^ " " ^ mdecl.mname ^
                            "= new Matrix(" ^ string_of_int mdecl.mrow ^ ", " ^ string_of_int mdecl.mcol ^ ")"
(*put struct id into the declaration list inside struct declaration*)
let rec tuple_id (id,nums) = match nums with
| [] -> []
| head :: tail -> (id,head):: tuple_id (id,tail)
(*string for expression*)
let rec string_of_expr_javagen = function
    Float_lit(l) -> string_of_float l
  | Int_lit(l) -> string_of_int l
  | String_lit(l) -> "\"" ^ l ^ "\""
  | Id(s) -> s
  | Binary_op(e1, o, e2) ->
      string_of_expr_javagen e1 ^ " " ^
      (match o with
      | (*And | Or | Eq | Neq | Lt | Gt | Leq | Geq*)
      Add -> "+" | Sub -> "-" | Times -> "*" | Divide -> "/" | Eq -> "=="
      | Neq -> "!="       | Lt -> "<" | Leq -> "<=" | Gt -> ">" | Geq -> ">="
      | And -> "&&" | Or -> "||"
      ) ^ " " ^
      string_of_expr_javagen e2
   (*matrix binary operations*)
  | MatBinary_op(e1, o, e2) ->
      (match o with
      MAdd -> "MatrixMathematics.add("^string_of_expr_javagen e1^" ,"^string_of_expr_javagen e2^")"
      | MSub -> "MatrixMathematics.subtract("^string_of_expr_javagen e1^" ,"^string_of_expr_javagen e2^")"
      | MTime -> "MatrixMathematics.multiply("^string_of_expr_javagen e1^" ,"^string_of_expr_javagen e2^")"
      | MDivide -> "MatrixMathematics.multiply("^string_of_expr_javagen e1^" ,"^"MatrixMathematics.inverse("^string_of_expr_javagen e2^"))"
      | MITime  ->  "("^string_of_expr_javagen e1^").multiplyByConstant("^string_of_expr_javagen e2^")"
      | MIDivide  ->  "("^string_of_expr_javagen e1^").multiplyByConstant(1.000/("^string_of_expr_javagen e2^"))"
      | MIAdd  ->  "("^string_of_expr_javagen e1^").addByConstant("^string_of_expr_javagen e2^")"
      | MISub  ->  "("^string_of_expr_javagen e1^").addByConstant(-1*("^string_of_expr_javagen e2^"))"
      )
   (*unitary operation*)
  | MatUnary_op(e, o) ->
    (match o with
    MTranspose -> "MatrixMathematics.transpose" | MInversion -> "MatrixMathematics.inverse" | MDeterminant -> "MatrixMathematics.determinant"
    ) ^ "(" ^ string_of_expr_javagen e ^ ")"
   (* assign codes *)
  | VarAssign(v, e) -> v ^ " = " ^ string_of_expr_javagen e
  | Matrix_element_assign (id,indexR,indexC,assignV)-> id^".data" ^"["^ string_of_expr_javagen indexR^"]["^ string_of_expr_javagen indexC^"] = "^string_of_expr_javagen assignV
  | Struct_element_assign (struct_id,member_id, expr) ->  struct_id^".valMap.put(\"" ^member_id^"\","^string_of_expr_javagen expr^");\n"
  | Matrix_element (v,e1,e2)->v^".data" ^"["^ string_of_expr_javagen e1^"]["^ string_of_expr_javagen e2^"] "
  (* functions that are built in are listed here*)
  | Call(f, el) -> (match f with
    | "printM" -> "("^String.concat ", " (List.map string_of_expr_javagen el)^").print()"
    | "toInt" -> "Integer.parseInt("^String.concat ", " (List.map string_of_expr_javagen  el)^")"
    | "toFloat" -> "Double.parseDouble("^String.concat ", " (List.map string_of_expr_javagen  el)^")"
    | "toBoolean" -> "Boolean.parseBoolean("^String.concat ", " (List.map string_of_expr_javagen  el)^")"
    | "toString" -> "ToString.toString(" ^ String.concat ", " (List.map string_of_expr_javagen  el)^")"
    | "print" -> "System.out.println(ToString.toString(" ^ String.concat ", " (List.map string_of_expr_javagen  el)^"))"
```

```
    | "price" -> "(" ^ String.concat ", " (List.map string_of_expr_javagen  el)^").price()"
    | "priceM" -> "Option.priceM(" ^ String.concat ", " (List.map string_of_expr_javagen  el)^")"
    |_  ->  f^"(" ^ String.concat ", " (List.map string_of_expr_javagen el) ^ ")"
    )
  | Noexpr -> "void"
  | Precedence_expr(e) -> "( " ^ string_of_expr_javagen e ^ " )"
  | Struct_element(struct_id, element_id) -> struct_id^".valMap" ^ ".get(\"" ^ element_id^"\")"
  | _ -> "space"
(*string for struct*)
let string_of_struct_arg (struct_id,arg) = struct_id^".valMap.put(\""^arg.id ^ "\" , " ^ string_of_expr_javagen arg.value^")"
(*string of global statement*)
let rec string_of_global_stmt = function
  Vardec(vdecl) -> string_of_global_vdecl vdecl ^ ";\n"
  | _  -> ""
(*string of statement*)
let rec string_of_stmt = function
    Block(stmts) -> "{\n  " ^ String.concat "  " (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr_javagen expr ^ ";\n"
  | Return(expr) -> "return " ^ string_of_expr_javagen expr ^ ";\n"
  (*| If(e, s, Block([])) -> "if (" ^ string_of_expr_javagen e ^ ")\n" ^ string_of_stmt s*)
  | If(e, s1, s2) ->  "if (" ^ string_of_expr_javagen e ^ ") " ^
      string_of_stmt s1 ^ "else " ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr_javagen e1  ^ " ; " ^ string_of_expr_javagen e2 ^ " ; " ^
      string_of_expr_javagen e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr_javagen e ^ ") " ^ string_of_stmt s
  | Vardec(vdecl) -> string_of_vdecl vdecl ^ ";\n"
  | Matdec(mdecl) -> string_of_mdecl mdecl ^ ";\n"
  | Structdec(struct_id, argList) -> "Structure "
  ^ struct_id ^ " = new Structure();\n" ^
   String.concat ";\n "
  (List.map (string_of_struct_arg ) (tuple_id (struct_id,(List.rev argList)) )) ^ ";\n"
  | Optiondec (struct_id, argList) -> "Option  "
  ^ struct_id ^ " = new Option();\n" ^
   String.concat ";\n "
  (List.map (string_of_struct_arg ) (tuple_id (struct_id,(List.rev argList)) )) ^ ";\n"
(*function decoration*)
let string_of_fdecl fdecl =
 "public static "^string_of_dataType fdecl.ret ^" " ^ fdecl.func_name ^ "("
^ String.concat ", " (List.map string_of_vdecl_argument fdecl.formals) ^ ")\n{\ntry{" ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  (* exception simply exits *)
  "}catch(Exception e){"^(match fdecl.ret with
| Int -> "return -1"
  | Float-> "return 0.0"
  | String -> "return null"
  | Matrix-> "return null"
  | Option -> "return null"
  | Structure -> "return null"
  | Boolean -> "return false"
  | Void  -> "return "
  )
  ^";}}\n"
(*convert ast into string*)
let string_of_program_gen (stmts, funcs)=
  String.concat "\n" (List.map string_of_global_stmt (List.rev stmts)) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl (List.rev funcs)) ^ "\n"
(*io function*)

let  writeToFile fileName progString =
  let file = open_out ("java/" ^ fileName ^ ".java") in
    let content=fprintf file "%s"  progString in
    "Write finished!"
(*generate program and output to file*)
let gen_program fileName prog = (*have a writetofile*)
  let programString = string_of_program_gen (fst prog ,snd prog ) in
  let out = sprintf
  "\npublic class %s\n{\n%s%s\n}" fileName programString "public static void main(String[] args){ try{main(0,\"\");}catch(Exception e){return ;}}"
in
    writeToFile fileName out
```

```
(* angelaZ compiler
  1. give stdin to scanner, get tokens
  2. give tokens to parser, get AST
  3. give AST to analyzer, get semantic tree
  4. give semantic tree to java converter, get java tree
  5. give java tree to java code generator, get java code
  6. give java code to java compiler, get executable
  7. run java executable
*)

type action =  Sast | Java

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [  ("-s", Sast); ("-j", Java)]
  else Java in
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  match action with
  Sast ->
      Typecheck.check_program program
  | Java ->
      let _ = Javagen.gen_program "Output" program in
      print_string "Success! Compiled to java/output.java\n"
```

```ocaml
open Printf
open Ast
(* data type string gen *)
let string_of_dataType = function
    Int -> "int"
  | Float -> "double"
  | String -> "String"
  | Matrix -> "Matrix"
  | Option -> "Option"
  | Structure -> "Structure"
  | Boolean -> "boolean"
  | Void -> "void"
(* global variable string gen *)
let string_of_vdecl vdecl =  string_of_dataType vdecl.vtype ^ " " ^ vdecl.vname ^ match vdecl.vtype with
| Int -> "=0"
  | Float-> "=0.0"
  | String -> "=null"
  | Matrix-> ""
  | Option -> ""
  | Structure -> ""
  | Boolean -> "=false"
  | Void  -> ""
(* variable declaration string gen notice the static *)
let string_of_global_vdecl vdecl =  "static "^string_of_dataType vdecl.vtype ^ " " ^ vdecl.vname ^ match vdecl.vtype with
| Int -> "=0"
  | Float-> "=0.0"
  | String -> "=null"
  | Matrix-> ""
  | Option -> ""
  | Structure -> ""
  | Boolean -> "=false"
  | Void  -> ""


let string_of_vdecl_argument vdecl =  string_of_dataType vdecl.vtype ^ " " ^ vdecl.vname
let string_of_mdecl mdecl = string_of_dataType mdecl.mtype ^ " " ^ mdecl.mname ^
                            "= new Matrix(" ^ string_of_int mdecl.mrow ^ ", " ^ string_of_int mdecl.mcol ^ ")"
(*put struct id into the declaration list inside struct declaration*)
let rec tuple_id (id,nums) = match nums with
| [] -> []
| head :: tail -> (id,head):: tuple_id (id,tail)
(*string for expression*)
let rec string_of_expr_javagen = function
    Float_lit(l) -> string_of_float l
  | Int_lit(l) -> string_of_int l
  | String_lit(l) -> "\"" ^ l ^ "\""
  | Id(s) -> s
  | Binary_op(e1, o, e2) ->
      string_of_expr_javagen e1 ^ " " ^
      (match o with
      | (*And | Or | Eq | Neq | Lt | Gt | Leq | Geq*)
      Add -> "+" | Sub -> "-" | Times -> "*" | Divide -> "/" | Eq -> "=="
      | Neq -> "!="       | Lt -> "<" | Leq -> "<=" | Gt -> ">" | Geq -> ">="
      | And -> "&&" | Or -> "||"
      ) ^ " " ^
      string_of_expr_javagen e2
   (*matrix binary operations*)
  | MatBinary_op(e1, o, e2) ->
      (match o with
      MAdd -> "MatrixMathematics.add("^string_of_expr_javagen e1^" ,"^string_of_expr_javagen e2^")"
      | MSub -> "MatrixMathematics.subtract("^string_of_expr_javagen e1^" ,"^string_of_expr_javagen e2^")"
      | MTime -> "MatrixMathematics.multiply("^string_of_expr_javagen e1^" ,"^string_of_expr_javagen e2^")"
      | MDivide -> "MatrixMathematics.multiply("^string_of_expr_javagen e1^" ,"^"MatrixMathematics.inverse("^string_of_expr_javagen e2^"))"
      | MITime  ->  "("^string_of_expr_javagen e1^").multiplyByConstant("^string_of_expr_javagen e2^")"
      | MIDivide  ->  "("^string_of_expr_javagen e1^").multiplyByConstant(1.000/("^string_of_expr_javagen e2^"))"
      | MIAdd  ->  "("^string_of_expr_javagen e1^").addByConstant("^string_of_expr_javagen e2^")"
      | MISub  ->  "("^string_of_expr_javagen e1^").addByConstant(-1*("^string_of_expr_javagen e2^"))"
      )
   (*unitary operation*)
  | MatUnary_op(e, o) ->
    (match o with
    MTranspose -> "MatrixMathematics.transpose" | MInversion -> "MatrixMathematics.inverse" | MDeterminant -> "MatrixMathematics.determinant"
    ) ^ "(" ^ string_of_expr_javagen e ^ ")"
   (* assign codes *)
  | VarAssign(v, e) -> v ^ " = " ^ string_of_expr_javagen e
  | Matrix_element_assign (id,indexR,indexC,assignV)-> id^".data" ^"["^ string_of_expr_javagen indexR^"]["^ string_of_expr_javagen indexC^"] = "^string_of_expr_javagen assignV
  | Struct_element_assign (struct_id,member_id, expr) ->  struct_id^".valMap.put(\"" ^member_id^"\","^string_of_expr_javagen expr^");\n"
  | Matrix_element (v,e1,e2)->v^".data" ^"["^ string_of_expr_javagen e1^"]["^ string_of_expr_javagen e2^"] "
  (* functions that are built in are listed here*)
  | Call(f, el) -> (match f with
    | "printM" -> "("^String.concat ", " (List.map string_of_expr_javagen el)^").print()"
    | "toInt" -> "Integer.parseInt("^String.concat ", " (List.map string_of_expr_javagen  el)^")"
    | "toFloat" -> "Double.parseDouble("^String.concat ", " (List.map string_of_expr_javagen  el)^")"
    | "toBoolean" -> "Boolean.parseBoolean("^String.concat ", " (List.map string_of_expr_javagen  el)^")"
    | "toString" -> "ToString.toString(" ^ String.concat ", " (List.map string_of_expr_javagen  el)^")"
    | "print" -> "System.out.println(ToString.toString(" ^ String.concat ", " (List.map string_of_expr_javagen  el)^"))"
    | "price" -> "(" ^ String.concat ", " (List.map string_of_expr_javagen  el)^").price()"
    | "priceM" -> "Option.priceM(" ^ String.concat ", " (List.map string_of_expr_javagen  el)^")"
```

```
    |_  ->  f^"(" ^ String.concat ", " (List.map string_of_expr_javagen el) ^ ")"
    )
  | Noexpr -> "void"
  | Precedence_expr(e) -> "( " ^ string_of_expr_javagen e ^ " )"
  | Struct_element(struct_id, element_id) -> struct_id^".valMap" ^ ".get(\"" ^ element_id^"\")"
  | _ -> "space"
(*string for struct*)
let string_of_struct_arg (struct_id,arg) = struct_id^".valMap.put(\""^arg.id ^ "\" , " ^ string_of_expr_javagen arg.value^")"
(*string of global statement*)
let rec string_of_global_stmt = function
  Vardec(vdecl) -> string_of_global_vdecl vdecl ^ ";\n"
  | _  -> ""
(*string of statement*)
let rec string_of_stmt = function
    Block(stmts) -> "{\n  " ^ String.concat "  " (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr_javagen expr ^ ";\n"
  | Return(expr) -> "return " ^ string_of_expr_javagen expr ^ ";\n"
  (*| If(e, s, Block([])) -> "if (" ^ string_of_expr_javagen e ^ ")\n" ^ string_of_stmt s*)
  | If(e, s1, s2) ->  "if (" ^ string_of_expr_javagen e ^ ") " ^
      string_of_stmt s1 ^ "else " ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr_javagen e1  ^ " ; " ^ string_of_expr_javagen e2 ^ " ; " ^
      string_of_expr_javagen e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr_javagen e ^ ") " ^ string_of_stmt s
  | Vardec(vdecl) -> string_of_vdecl vdecl ^ ";\n"
  | Matdec(mdecl) -> string_of_mdecl mdecl ^ ";\n"
  | Structdec(struct_id, argList) -> "Structure "
  ^ struct_id ^ " = new Structure();\n" ^
   String.concat ";\n "
  (List.map (string_of_struct_arg ) (tuple_id (struct_id,(List.rev argList)) )) ^ ";\n"
  | Optiondec (struct_id, argList) -> "Option  "
  ^ struct_id ^ " = new Option();\n" ^
   String.concat ";\n "
  (List.map (string_of_struct_arg ) (tuple_id (struct_id,(List.rev argList)) )) ^ ";\n"
(*function decoration*)
let string_of_fdecl fdecl =
 "public static "^string_of_dataType fdecl.ret ^" " ^ fdecl.func_name ^ "("
^ String.concat ", " (List.map string_of_vdecl_argument fdecl.formals) ^ ")\n{\ntry{" ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  (* exception simply exits *)
  "}catch(Exception e){"^(match fdecl.ret with
| Int -> "return -1"
  | Float-> "return 0.0"
  | String -> "return null"
  | Matrix-> "return null"
  | Option -> "return null"
  | Structure -> "return null"
  | Boolean -> "return false"
  | Void  -> "return "
  )
  ^";}}\n"
(*convert ast into string*)
let string_of_program_gen (stmts, funcs)=
  String.concat "\n" (List.map string_of_global_stmt (List.rev stmts)) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl (List.rev funcs)) ^ "\n"
(*io function*)

let  writeToFile fileName progString =
  let file = open_out ("java/" ^ fileName ^ ".java") in
    let content=fprintf file "%s"  progString in
    "Write finished!"
(*generate program and output to file*)
let gen_program fileName prog = (*have a writetofile*)
  let programString = string_of_program_gen (fst prog ,snd prog ) in
  programString
  (*let out = sprintf
  "\npublic class %s\n{\n%s%s\n}" fileName programString "public static void main(String[] args){ try{main(0,\"\");}catch(Exception e){return ;}}"
in
    writeToFile fileName out*)
```

```
(* angelaZ compiler
  1. give stdin to scanner, get tokens
  2. give tokens to parser, get AST
  3. give AST to analyzer, get semantic tree
  4. give semantic tree to java converter, get java tree
  5. give java tree to java code generator, get java code
  6. give java code to java compiler, get executable
  7. run java executable
*)

type action =  Sast | Java

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [  ("-s", Sast); ("-j", Java)]
  else Java in
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  match action with
  Sast ->
      let result = Typecheck.check_program program in
      ()
  | Java ->
      let result = JavagenTest.gen_program "Output" program in
      print_string result
```

```
Int i;
Boolean b;

Matrix main2(Int argc, String argv) {
    Matrix m3(2,2);
    Matrix m(2,2);
    m[0][0]=1;
    m[0][1]=2;
    m[1][0]=3;
    m[1][1]=4;
    m3 = (((m +. m') *. m~) *.. 4)+.. m^;
    return m3;
}

Void main(Int argc2, String m) {
    Matrix result(2,2);
    result=main2(0, "str");
    printM(result);
}
```

```
Int i;
Boolean b;

Structure main2(Int argc, String argv) {
    Structure s={a="1", b= toString(argc)};
    i=toInt(s -> a);
    return s;
}

Void main(Int argc2, String m) {
    Structure result={};
    result=main2(0, "str");
    print(result);
}
```

```
Float i;
{*
Here is the comment you don't care about!
*}
Option main2(Int argc, String argv) {
    Option s={strike="100.0", stock= "150.0", interestRate="0.1", period="1.0", sigma="2.0", optionType="call"};
    i=toFloat(s -> strike);
    return s;
}
Matrix main3(Int a) {
    Matrix strike(1,2);
    strike[0][0]=10;
    strike[0][1]=20;
    Matrix stock(1,2);
    stock[0][0]=15;
    stock[0][1]=25;
    Matrix interestRate(1,2);
    interestRate[0][0]=0.4;
    interestRate[0][1]=0.1;
    Matrix period(1,2);
    period[0][0]=3;
    period[0][1]=4;
    Matrix sigma(1,2);
    sigma[0][0]=0.1;
    sigma[0][1]=0.2;

    Matrix s(1,1);
    s= priceM(strike,stock,interestRate,period,sigma);
    return s;
}
Void main(Int argc2, String m) {
    Option result={};
    result=main2(0, "str");
    Float d;
    d=price(result);
    print(d);
    Matrix result1(1,1);
    result1=main3(0);
    print(result1);
}
```

```
OBJS = ast.cmo sast.cmo parser.cmo scanner.cmo typecheck.cmo javagen.cmo angelaZ.cmo javagenTest.cmo angelaZTest.cmo
#calc.cmo
OBJS_TEST = ast.cmo sast.cmo parser.cmo scanner.cmo typecheck.cmo javagenTest.cmo angelaZTest.cmo

angelaZ: $(OBJS)
	ocamlc -g -o angelaZ $(OBJS)
#calc : $(OBJS)
#	ocamlc -o calc $(OBJS)
angelaZTest: $(OBJS_TEST)
	ocamlc -g -o angelaZTest $(OBJS_TEST)
.PHONY : test
test : angelaZTest testall.sh
	./testall.sh

scanner.ml : scanner.mll
	ocamllex scanner.mll

parser.ml parser.mli : parser.mly
	ocamlyacc parser.mly

%.cmo : %.ml
	ocamlc -w  -A -c $<

%.cmi : %.mli
	ocamlc -w -A -c $<

#calc.tar.gz : $(TARFILES)
#	cd .. && tar czf calc/calc.tar.gz $(TARFILES:%=calc/%)

.PHONY : clean
clean :
	rm -f calc parser.ml parser.mli scanner.ml testall.log \
	*.cmo *.cmi *.out *.diff

# Generated by ocamldep *.ml *.mli
typecheck.cmo : sast.cmo ast.cmo
typecheck.cmx : sast.cmx ast.cmx
ast.cmo:
ast.cmx:
#calc.cmo: scanner.cmo parser.cmi ast.cmo
#calc.cmx: scanner.cmx parser.cmx ast.cmx
parser.cmo: ast.cmo parser.cmi
parser.cmx: ast.cmx parser.cmi
sast.cmo : ast.cmo
sast.cmx : ast.cmx
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
parser.cmi: ast.cmo
javagen.cmo : ast.cmo
javagen.cmx : ast.cmx
javagenTest.cmo : ast.cmo
javagenTest.cmx : ast.cmx
pc.cmo : scanner.cmo  parser.cmi javagen.cmo ast.cmo javagenTest.cmo
pc.cmx : scanner.cmx  parser.cmx javagen.cmx ast.cmx javagenTest.cmx
```

```sh
#!/bin/sh

MICROC="./angelaZTest"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.mc files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
    echo "FAILED"
    error=1
    fi
    echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
    SignalError "$1 $2 differs"
    echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
    SignalError "$1 failed on $*"
    return 1
    }
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.mc//'`
    reffile=`echo $1 | sed 's/.mc$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

#    generatedfiles="$generatedfiles ${basename}.i.out" &&
#Run "$MICROC"  "<" $1 ">" ${basename}.i.out &&
#Compare ${basename}.i.out ${reffile}.out ${basename}.i.diff

    generatedfiles="$generatedfiles ${basename}.c.out" &&
    Run "$MICROC"  "<" $1 ">" ${basename}.c.out  &&
    Compare ${basename}.c.out ${reffile}.out ${basename}.c.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
    if [ $keep -eq 0 ] ; then
        rm -f $generatedfiles
    fi
    echo "OK"
    echo "###### SUCCESS" 1>&2
    else
    echo "###### FAILED" 1>&2
    globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
    k) # Keep intermediate files
```

```
        keep=1
        ;;
    h) # Help
        Usage
        ;;
    esac
done

shift `expr $OPTIND - 1`

if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/fail-*.az tests/test-*.az"
fi

for file in $files
do
    case $file in
    *test-*)
        Check $file 2>> $globallog
        ;;
    *fail-*)
        CheckFail $file 2>> $globallog
        ;;
    *)
        echo "unknown file type $file"
        globalerror=1
        ;;
    esac
done

exit $globalerror
```