Angela-Z: A Language that facilitate the Matrix-wise operations
Language Reference Manual

Fei Liu, Mengdi Zhang, Taikun Liu, Jiayi Yan

Contents

1. Language definition

1.1. Usage

Angela-Z is a language designed for computing and tracing option pricings. It can also be used as a general-purpose programming language.

1.2. What special feature do we have?

We support the built-in Matrix and Option class. Matrix-wise operations such as Plus, Minus, Multiply, and Division are all supported. Direct matrix declaration, operations, and access are supported with minimum effort. Option is provided for option pricing computing. The language should be able to shelter the user from intensive loops and manipulation of matrices or collection of option pricing data.

1.3. Example:

```
{**The following piece code express a LU decomposition algorithm**}
float s;
matrix U;
matrix L;
matrix PV;
s=size(A);
U=A;
L(s,s);
PV=transpose(A[0 to s-1]);
for( float j=1;j<s;j++) {
   float ind;
   ind=max(abs(U[j to s, j]));
   ind=ind+j-1;
   matrix t;
   t=PV[j]; PV[j]=PV[ind]; PV[ind]=t;
   t=L[j,1 to j-1]; L[j,1 to j-1]=L[ind,1 to j-1]; L[ind,1 to j-1]=t;
   t=U[j,j to end]; U[j,j to end]=U[ind,j to end]; U[ind,j to end]=t;
   L[j,j]=1;
   for( float i=(1+j);i< size(U,1);i++) {
      float c;
      c= U[i,j]/U[j,j];
      U[i,j to s]=U[i,j to s]-U[j,j to s]*c;
      L[i,j]=c;
   }
}
```

2. Lexical Conventions

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++.

2.2. Identifiers

An identifier is a combination of a series of characters, which include letters, digits as well as underscore'_'. The identifiers are case sensitive which mean that 'Foo' is not the same as 'foo'. And it can only start with letters.

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

2.4. Literals Types

2.4.1. Literal primitive types

The following table contains examples of literal primitive types.

integer	0 1 99
float	0.0 0.002 2e-3 99.0
string	“Hello World”
boolean	true false

Integer
An integer consists of a sequence of digits and no a decimal point. 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 optionally signed integer exponent.

String
A string is enclosed in double quotation, for example "Hello World".

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

2.4.2. Literal collection type

Matrix
A matrix data type consists rows and columns of floats. The elements in the matrix can be accessed by its index, for example m[1,2] can access the element in the first row and second column of matrix m.

2.5. 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

Operators for Matrix

Symbol	Explanation
+.	Performs addition
-.	Performs minus
*.	Performs multiplication
/.	Performs division
=	Performs assignment
(,)	Performs initialization
[,]	Performs index accessing

Operators for Option, Structure

Symbol	Explanation
=	Performs assignment
(:)	Performs initialization
->	Performs element 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

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

2.6. 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

3. Meaning of Identifiers

3.1. Object Types

Our language supports the four fundamental types of objects:

- integer
- floating point

- string
- boolean

And we supports advanced types of objects:

- Matrix
- Structure
- Option

Character Types

The only supported character type is the string type. This 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.

Integer and Floating Point types

The only supported integer type is int and the only supported floating point type is float. Both of these data types are signed.

Boolean Type

This type is a truth value and can only store a bit of information. Specifically, it may only take a value of either true or false.

Matrix Type

This type only supports two dimensional matrix. 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. This can only store float point type values.

Structure Type

This type is kind like map type. It can store pairs of (key, value). There is no upper bound for storing key-value pairs. Key must be a string type and value must be a float type.

Option Type

This type is an extension of Structure Type. It has some built-in functions and preset (key, value) pairs.

4. Type Conversions

No type conversion is allowed.

5. Expressions and Operators

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. There is no parentheses to force precedence in the language.

Initialization, i.e., Structure or Option initialization (:) and Matrix initialization (,), 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	Left
[,]	Matrix element accessing	Non-associative
* / *. /.	Times, Divide, Matrix times, Matrix divide	Left
+ - +. -.	Plus, Minus, Matrix plus, Matrix minus	Left
< <= > >=	Less than, Less than or equal to, Larger than, Larger than or equal to	Left
== != && ||	Equality, Inequality, Logical AND, Logical OR	Left
=	Assignment	Right

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 Section 2.4 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.

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 *}
```

5.4. Additive Operators

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

- Plus (+)
- Minus (-)
- Matrix Plus (+.)
- Matrix Minux (-.)

Types used with the first two additive operators are Int and 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. Note that operands before and after operators should of same type. Plus or minus of an Int and a Float is not supported.

Examples are as follows:

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

Types used with the last 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(2,3,4);
Matrix b(1,2,3);
Matrix c;
c = a - b;
{* c is a matrix of size 1*3: 1,1,1*}
```

5.5. Multiplicative Operators

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

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

Types used with the first two multiplicative operators are Int and 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. Note that operands before and after operators should of same type. Multiplication or division of an Int and a Float is not supported.

Examples are as follows:

```
Float a;
Float b;
a = 1.1;
b = a * 0.2;
{* b = 2.2*}
```

Types used with the last 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. All elements in the matrix are treated as Float.

Examples are as follows:

```
Matrix a(2,0);
Matrix b((1,2),(0,1));
Matrix c;
c = a *. b;
{* c is a matrix of size 1*2: 2,0*}
```

5.6. 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 (0) if the relationship in the expression is false; otherwise, the value returned is true (1). Relation comparison of an Int and a Float is not supported.

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 gives an error since a is Float and b is Int
*}
```

5.7. 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 (1) if both operands have the same value; otherwise, it returns false (0). The not-equal-to operator (!=) returns true if the operands do not have the same value; otherwise, it returns false. Equality comparison of two 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
*}
```

5.8. Boolean Operators

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

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

Types used with the Boolean operators are Boolean. The logical AND operator (&&) returns the Boolean value true if both operands are true and returns false otherwise. The operands are implicitly converted to type Boolean prior to evaluation.

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
*}
```

5.9. 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.

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
*}
```

5.10. Initialization Operators

The Initialization Operators have no associativity:

- Structure or Option initialization ((:))
- Matrix initialization ((,))

Types used with the first initialization operator are Structure and Option. It should be of format Type Id(key:value, key:value….). Note that trying to access a Structure or Option that hasn't been initialized causes error.

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. A row is represented as (, , ...). A matrix with one row is initialized in the format Type Id(value, value,...). A matrix with several rows is initialized in the format Type Id(value, value,...).Note that trying to access a Structure or Option that hasn't been initialized causes error.

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") *}
```

5.11. 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 object in the index position indicated by the int number after the operator. Index 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:

```
Option a(id:5, country: "China");
if ( a->1 == "China")
   {...}
{* a->1 returns "China" and so a->1 == "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]. 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(1,2,3);
Int b;
b = a[0,2]
{* b = 3 *}
```

6. Declarations

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) { … }
```

6.2. Variable Declarations

Variables are declared in the following way:

```
dataType  varName;
```

dataType could be int, float, string, boolean, void, matrix, option, option. varName must be a valid identifier that is a combination of characters, and can only begin with letters.

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;
```

7. Statements

7.1. Block

A block encloses a series of statements by braces.

```
{
```

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

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.

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.

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.

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.

8. System Functions

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! *}
```