# Uniform General Algorithmic (UNIGA)

## *Financial Trading Language*

# Language Reference Manual


Leon Wu (llw2107@columbia.edu)
Jiahua Ni (jn2173@columbia.edu)
Jian Pan (jp2472@columbia.edu)
Yang Sha (ys2280@columbia.edu)
Yu Song (ys2310@columbia.edu)

**Columbia University in the City of New York**

# Introduction

Today many financial firms have their own trading software tools to facilitate investors' investment process. These tools often differ from each other and usually take a long time to learn. They are also not easy to be customized and very expensive. Many institutional and individual investors have their own custom-designed investment strategies. They want to implement their investment strategies using some easy-to-use software with low cost. However, designing financial trading software from scratch requires professional knowledge and can be costly and time-consuming. The UNIGA Language provides an easy and efficient way for people to design their own investment plans. The language allows users to write a program that can automatically trade financial instruments including Stock, Options, Bonds, and Mutual Funds etc. using pre-defined trading strategy that defines trading rules such as set-price, sell-price, comparisons, quantity and other elements.

# Overview of this Language

UNIGA is a high-level scripting language. Script programming languages enable programmers to specify trading operations intuitively. Although not as comprehensive as the more well known scripting languages such as Perl or Python, the built-in keywords make the language more intuitive and easy to use. The user is able to design trading software in the form of a program. The translator will then output a Java source file that can be edited and compiled into Java byte-code.

# 1. Lexicon

## 1.1 Identifiers

Identifiers consist of a sequence of one or more uppercase or lowercase alphabetic characters, (digits 0 to 9). The first character of an identifier should be a letter and cannot be a number. Identifiers are case sensitive, upper case and lower case letters are treated differently. Keywords are not identifiers.

## 1.2 Keywords

The following identifiers are reserved as keywords:

| | | | |
|---|---|---|---|
| boolean | double | date | void |
| if | else | for | break |
| while | function | return | market |
| open | close | volume | high |
| low | sell | buy | |

## 1.3 Numbers

A number consists of digits, decimal point "." All numbers in our language are implemented as double precision floating point numbers by default.

## 1.4 String literals

Strings are sequences of zero or more characters. String literals are character strings surrounded by quotation marks (“ ”). String literals can include any valid character, including white-space characters and character escape sequences.

### 1.5 Operators

An operator is a token that specifies an operation on at least one operand, and yields some result (a value, designator, side effect, or some combination). Operands are expressions or constants (a form of expression).

Operators in UNIGA are:

| | | | | | |
|---|---|---|---|---|---|
| + | - | * | / | | |
| > | < | = | = = | & | \| |

## 2. Data Types

We have defined our data type as follows:
Numeric: double (to represent date, time, price, trade volume), and its array double[];
True/False: Boolean values, there’s no array for this data type.

## 3. Declarations

### 3.1 General type declaration

The general syntax of a declaration is as follows:

**declaration:**
*type-specifier init-declarator-list(opt);*

**init-declarator-list:**
*declarator*
*init_declarator-list , declarator*

Type specifiers are: double, boolean

### 3.2 Declaring Arrays

Arrays are declared with the bracket punctuators [ ], as shown in the following syntax:

*type-specifier declarator [constant-expression-list(opt)]*

## 4. Functions

### 4.1 Function Calls

A *function call* is a primary expression, usually a function identifier followed by parentheses, which is used to invoke a function. The parentheses contain a (possibly empty) comma-separated list of expressions that are the arguments to the function.

## 4.2 Functions Types

A function has the derived type "function returning *type*". The *type* can be any data type except array types or function types. If the function returns no value, its type is "function returning void ", sometimes called a *void function.*

Functions can be introduced into a program in one of two ways:
1. A *function definition* can create a function designator, define its parameters and their type, define the type of its return value, and supply the body of the function.
2. A *function declaration* announces the properties of a function defined elsewhere.

## 4.3 Function Definitions

A function definition includes the code for the function. Function definitions can appear in any order, and in one source file or several, although a function cannot be split between files. Function definitions cannot be nested.

A function definition has the following syntax:

**function-definition:**
*function declaration-specifiers(opt) declarator declaration-list(opt)*
*compound-statement*

**declaration-specifiers**
*The declaration-specifiers (type-qualifier, and type- specifier) can be listed in any order.*

Type specifiers are: double, boolean

Type qualifiers are: const

***Example:***

```
main () {
    buy "MSFT" 100;
    buy "MSFT" Max(100, 50);
}

function double Max( x, y ) {
    if  x > y then
        return x;
    else
        return y;
}
```

## 4.4 Function Declarations

For all functions if the function definition is located after the calling function in the source code, the function must be declared before calling it.

## 4.5 Function Parameters and Arguments

UNIGA functions exchange information by means of parameters and arguments. The term parameter refers to any declaration within the parentheses following the function name in a function declaration or definition; the term argument refers to any expression within the parentheses of a function call.

The following rules apply to parameters and arguments of UNIGA functions:

- Except for functions with variable-length argument lists, the number of arguments in a function call must be the same as the number of parameters in the function definition. This number can be zero.
- The maximum number of arguments (and corresponding parameters) is 50 for a single function.
- Arguments are separated by commas. However, the comma is not an operator in this context, and the arguments can be evaluated by the compiler in any order. There is, however, a sequence point before the actual call.
- Arguments are passed by value; that is, when a function is called, the parameter receives a copy of the argument's value, not its address. This rule applies to all scalar values, structures, and unions passed as arguments.
- Modifying a parameter does not modify the corresponding argument passed by the function call.

## 4.6 Function invocation and return

**Function call**

A function call can be a single statement followed by a “;”

**Return statement**

The return statement is used to return from the function at the point the return statement is specified Return statement can also return a value. The return statement is ended by a “;”.

## 4.7 Built-In functions

### 4.7.1 print()
Console output functions

### 4.7.2 load()
File I/O functions

### 4.7.3 error()
Standard error handler

### 4.7.4 stop()
Stop price

### 4.7.5 limit()

Sets the trading price limits

4.7.6 range()
Range of month or year

4.7.7 average()
Average price

4.7.8 sum()
Total amount

4.7.9 pl()
Profit and loss

# 5. Expressions and Operators

## 5.1 Primary expressions

Simple expressions are called primary expressions; they denote values. Primary expressions include previously declared identifiers, constants, string literals, and parenthesized expressions.

Primary expressions have the following syntax:

**primary-expression:**
*identifier*
*constant*
*string-literal*
*parenthesized expression*

### 5.1.1 Identifier
An identifier is a primary expression provided it is declared as designating an object or a function.

### 5.1.2 Constant
A constant is a primary expression. Its type depends on its form (in UNIGA, it's Boolean or double)

### 5.1.3 String Literals
A string literal is a primary expression

### 5.1.4 Parenthesized Expressions
An expression within parentheses has the same type and value as the expression without parentheses would have. Any expression can be delimited by parentheses to change the precedence of its operators.

## 5.2 Postfix Operators

Postfix expressions include array references, function calls and postfix increment and decrement expressions. The operators in postfix expressions have left-to-right associativity.

Postfix expressions have the following syntax:

**postfix-expression:**
*array-reference*
*function-call*

### 5.2.1 Array References

The bracket operator [ ] is used to refer to an element of an array. Array references have the following syntax:

**array-reference:**
*postfix-expression [ expression ]*

### 5.2.2 Function Calls

Function calls have the following syntax:

**function-call:**
*postfix-expression ( argument-expression-list(opt) )*

**argument-expression-list(opt):**
*assignment-expression*
*argument-expression-list(opt), assignment-expression*

A function call is a postfix expression consisting of a function designator followed by parentheses. The order of evaluation of any expressions in the function parameter list is undefined, but there is a sequence point before the actual call. The parentheses can contain a list of arguments (separated by commas) or can be empty.

## 5.3 Unary Operators

Unary expressions are formed by combining a unary operator with a single operand. All unary operators are of equal precedence and have right-to-left associativity. The unary operator is: Unary minus (-)

## 5.4. Binary Operators

The binary operators are categorized as follows:

- Multiplicative operators: multiplication (*), and division (/)
- Additive operators: addition (+) and subtraction (-)
- Relational operators: less than (<), greater than (>)
- Equality operators: equality (==) and inequality (!=)
- Logical operators: AND (&) and OR (|)

### 5.4.1 Multiplicative operators:

The multiplicative operators are *, /. Operands must have arithmetic type. Operands are converted, if necessary, according to the usual arithmetic conversion rules.

*The * operator performs multiplication.*

*The / operator performs division.*

### 5.4.2 Additive operators:
The additive operators + and - perform addition and subtraction. Operands are converted, if necessary, according to the usual arithmetic conversion rules.

### 5.4.3 Relational operators:
The relational operators compare two operands and produce a result of type int. The result is 0 if the relation is false, and 1 if it is true. The operators are: less than (<), greater than (>). Both operands must have an arithmetic type.

The relational operators associate from left to right.

### 5.4.4 Equality operators:
The equality operators, equal (==) and not-equal (!=), produce a result of type double , so that the result of the following statement is 1 if both operands have the same value, and 0 if they do not:

### 5.4.5 Logical operators:
The logical operators are AND (&) and OR (|). These operators guarantee left-to-right evaluation. The result of the expression (of type double ) is either 0 (false) or 1 (true). The operands need not have the same type, but both types must be scalar. If the compiler can make an evaluation by examining only the left operand, the right operand is not evaluated.

### 5.5 Assignment Operators

Assignments result in the value of the target variable after the assignment. They can be used as sub-expressions in larger expressions.

Assignment expressions have two operands: a modifiable value on the left and an expression on the right. A simple assignment consists of the equal sign (=) between two operands:

*Exp1 = Exp2;*

The value of expression Exp2 is assigned to Exp1. The type is the type of Exp1, and the result is the value of Exp1 after completion of the operation.

## 6. Statements

This section describes the following kinds of statements in the UNIGA programming language. Statements are executed in the sequence in which they appear in a function body. UNIGA supports the following types of statements:

- Compound statements
- Expression statements
- Null statements
- Selection statements
- Iteration statements
- Break statements
- Trading statements

### 6.1 Compound Statements

A compound statement, or block, allows a sequence of statements to be treated as a single statement. A compound statement begins with a left brace, contains optional declarations followed optionally by statements, and ends with a right brace, as shown in the following example:

**compound-statement:**
*{declaration-list}? {statement-list}?*

**declaration-list:**
*declaration*
*| declaration-list declaration*

**statement-list:**
*statement*
*| statement-list statement*

***Example:***

```
{
     x=2;
     y=3;

     if (x > y) then
          return x;
     else
          return y;
}
```

Block declarations are local to the block, and, for the rest of the block, they supersede other declarations of the same name in outer scopes.

## 6.2 Null Statement

A null statement is used to provide a null operation in situations where the grammar of the language requires a statement, but the program requires no work to be done. The null statement consists of a semicolon:

;

The null statement is useful with the if, while, and for statements. The most common use of this statement is in loop operations in which all the loop activity is performed by the test portion of the loop.

## 6.3 Selection Statement

### The if Statement

The if statement has the following syntax:

*if expression*

*then*
*statement*

*else(opt)*

*else-statement(opt)*

The statement following the control expression is executed if the value of the control expression is true (non-zero). An if statement can be written with an optional else clause that is executed if the control expression is false (0).

## 6.4 Iteration Statement

An iteration statement, or *loop*, repeatedly executes a statement, known as the *loop body*, until the controlling expression is false (0). The control expression must have a scalar type.

Iteration statement in UNIGA includes the following:

- The while statement evaluates the control expression before executing the loop body
- The for statement executes the loop body based on the evaluation of the second of three expressions

### 6.4.1 The while Statement

The while statement evaluates a control expression before each execution of the loop body. If the control expression is true (non-zero), the loop body is executed. If the control expression is false (0), the while statement terminates. The while statement has the following syntax:

*while ( expression )*

*statement*

### 6.4.2 The for Statement

The for statement evaluates three expressions and executes the loop body until the second controlling expression evaluates to false (0). The for statement is useful for executing a loop body a specified number of times. The for statement has the following syntax:

*for* ( *expression-1(opt)* ;

*expression-2(opt)* ; *expression-3(opt)*)
*statement*

The for statement executes the loop body zero or more times. Semicolons (;) are used to separate the control expressions. A for statement executes the following steps:

1. *expression-1* is evaluated once before the first iteration of the loop. This expression usually specifies the initial values for variables used in the loop.
2. *expression-2* is any scalar expression that determines whether to terminate the loop. *expression-2* is evaluated before each loop iteration. If the expression is true (nonzero), the loop body is executed. If the expression is false (0), execution of the for statement terminates.
3. *expression-3* is evaluated after each iteration.
4. The for statement executes until *expression-2* is false (0), or until a jump statement, such as break or goto, terminates execution of the loop.

## 6.5 Break Statement

The break statement causes the termination of the enclosing while and for. The control passes to the statement following the terminated statement.

### 6.6 Return Statement

The return statement cause program control to return to the caller. An optional expression following the return keyword will cause the function to return the value of the expression to the caller. If required, the expression is converted, as if by assignment, to the type of the function in which it appears.

### 6.7 Trading Statement

UNIGA provides standard trading statements which are easy to use.

*buy/sell stock_identifier expression1 expression2*

*stock_identifier* is of type string, and is generally the trading stock identifier. *expression1* is used to refer the volume to buy/sell. *expression2* is used to refer the price at which to buy/sell.

## 7. References

[1] ***C Reference Manual***, Dennis M.Ritchie