STL Reference Manual

the **S**tock **T**rading **L**anguage

Rekha Duthulur
Matt (Yu-Ming) Chang
Rui Hu
Nikhil Jhawar
Tom Lippincott

1. Introduction

STL is an interpreted computer language designed to facilitate the rapid prototyping of stock trading strategies. It is aimed at individuals in the financial industry, with little or no formal background in computer science. It therefore strives to use the industry's terminology whenever possible, and implement only the most useful and intuitive control structures. It borrows from Perl's accessible looping and variable conventions, and allows a similar flexibility in terms of overloaded operators and keywords. With the limited scope of financial simulation, defining "the right thing to do" in such situations is less problematic than for a general-purpose programming language.

1.1 Notation

The manual indicates syntactic categories with *italic* script and keywords with **bold** typeface. When there are several options for an item, they are separated by pipes (e.g. A|B|C).

2. Lexical Conventions

2.1 Comments

Like the C convention, comments begin with "/*" and end with "*/" and may continue over multiple lines.

2.2 Identifiers

Only three variable types are permitted in STL, each with a distinctive first character that clearly identifies the type at a glance:

2.21 Numerical quantities may be stored in variables starting with a dollar ($) sign,

2.22 Strings (typically stock names, in this context) may be stored in variables starting with a percentage (%) sign.

2.23 Lists may be stored in variables starting with an “at” (@) sign.

2.3 Keywords

The following words are reserved (see section 8 for definitions):

buy	**sell**
wma	**foreach**
while	**if**
else	**elsif**
wait	**[$|%|@]function**
return	**print**
delta	

2.4 Constants

2.41 A number is specified as one or more digits, with an optional decimal portion. A decimal portion must be preceded with a number (so “0.012”, not “.012”).

2.42 A string is specified as any combination of ascii characters enclosed within double quotes (“). To include a quotation mark in the string, it must be escaped with a preceding backslash (\). The string literal terminates with the first non-escaped matching quotation mark.

2.43 Lists are specified as comma-separated elements between straight brackets ([ ]). The elements must be either all strings/string variables or all numbers/number variables.

2.5 Whitespace

Whitespace is ignored by the STL compiler. The purpose of whitespace is to separate out different tokens and allow users to follow the code in an easier fashion. Whitespace includes indentations, tabs, spaces and line terminators (including support for DOS, UNIX and MAC standards).

2.6 Separators

The following characters are used in STL as separators:
{} – Code block separator
() – Grouping separator and parameter list separator
; - Statement Delimiter

3. Expressions

The following section discusses expressions used in STL. Expression operators are listed in order of highest to lowest level of precedence. Operators in the same section are considered to have

equal precedence and evaluate from left to right. Using any operator on a type without an implicit cast defined (i.e. multiplication on two strings) will result in a compile time error.

3.1 Primary Expressions

A primary expression is the simplest form of an expression and is typically used to represent a single value. Primary expressions include identifiers, constants and function calls. Furthermore, parenthesized expressions are considered to be primary expressions. Primary expressions are of highest precedence in STL

3.2 Unary Expressions

The STL compiler allows only a single unary operator which is the NOT operator. The result of a unary expression is the logical negation of the expression. Thus, the negation returns a one when the expression returns a zero and returns a zero when the expression is non-zero.

3.3 Multiplicative Operators

The two multiplicative operators: * for multiplication, / for division can be used to perform multiplicative operations. They are grouped left to right and can be applied to numerical constants.

3.4 Additive Operators

The two additive operators, + for addition and – for subtraction, can be used to perform additive operations. They are grouped left to right and can be applied to numerical data types.

3.5 Assignment Operators

The assignment operator, "=", stores the value returned by the expression on its right side into the identifier on its left side. It has the lowest precedence, and is associative from right to left. This allows multiple assignments to be made on the same line, in the following manner:

identifier1 = identifier2 = identifier3 = expression;

In this statement, the low precedence and right to left associatively of the assignment operator ensures that the entire expression is evaluated first. Next, the value of the expression is assigned to identifier3, then identifier2, then identifier1.

3.6 Relational and Equality Expressions

The following table summarizes the 6 relational operators available.

Operator	Description
>	Greater Than
<	Less Than

>=	Greater Than or Equal To
<=	Less Than or Equal To
==	Equal To
<>	Not Equal To

Relational and equality expressions are constructed as follows:

expression operator expression

Relational expressions return either zero or one, indicating whether the expression was false or true, respectively.

3.7 Logical Expressions

There are two logical operators available in STL. The logical AND operator, takes higher precedence than the logical inclusive OR operator. These operators are generally applied to relational and equality expressions, and other logical expressions. The values are evaluated, and the logical expression returns true or false (1 or 0, respectively). Logical expressions may be grouped with parenthesis like any other binary operation. The exclusive OR operator is not defined. Logical expressions treat zero as false, and non-zero values as true.

4. Operator Precedence

The following table shows the precedence of operators in STL from highest to lowest:

Operators	Associativity
(*expression*)	Left to Right
NOT	Right to Left
*, /	Left to Right
+, -	Left to Right
<, <=, >=, >, ==, <>	Left to Right
AND	Left to Right
OR	Left to Right
=	Right to Left

5. Function Declarations

User defined function has the following format:

[$|%|@] **function** function_name (%a, @b)
{
statement-list
}

The character preceding the function keyword indicates the return type. A function may only have return statements of this type: upon terminating without a return value, a function returns its type's default value (0, “” and [] respectively).

6. Statements

Except as indicated, statements are executed in sequence

6.1 Expression statement
Most statements are expression statements, which have the form
expression ;

6.2 Block
{*statement-list*}

6.3 Conditional statement
if (*expression*) *statement*
if (*expression*) *statement* **else** *statement*

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the **else** ambiguity is resolved by connecting an **else** with the last encountered **elsel**ess **if**

6.4 **while**(*expression*) { *statement* }

Continually execute *statement-list* until *expression* evaluates to false (0).

6.5 **foreach** *identifier* (*list*) { *statement* }

Iterate *statement-list* over the *list*, placing the current iteration's element in *identifier*.

6.6 **wait**(*expression*);

Pause execution until *expression* evaluates to true (1).

6.7 **return** *expression*;

In a function scope, exit the function and returns the value that *expression* evaluates to. Outside of a function scope, terminate the current program.

7. Scope rules

Variables defined in a function have the function body as its scope. The scope for variable defined outside of a function extends from their definition through the end of the file in which they appear.

8. Keywords

The distinction between functions (user defined, standard library, etc) and keywords is that the latter are immutable. The reasoning is that these operations are so integral to the domain (stock trading) that they may never be overwritten, and require no external definitions.

8.1 **buy** *number string|list*

purchase *number* of stock *string*, or *number* of each stock in *list*

8.2 **sell** *number string|list*
sell *number* of stock *string*, or *number* of each stock in *list*

8.3 **wma** *string|list*
return the weighted market average of stock *string*, or of all the stocks in *list*

8.4 **delta** *number string|list*
return the price change of stock *string*, or of all stocks in *list*, over the previous *number* days

8.5 **print** *string|list|number*
print out arguments appropriately (string, digits, or list thereof)

9. Standard Library

In contrast to keywords, the standard library defines common, industry-standard algorithms that a user might wish to override. This set of algorithms is unpopulated at the time of our language definition, but candidates will be considered based on their utility, standardization and ubiquity.