Traders Joe

Language Reference Manual Version 1.0

By Monu Chacko Email: mc3574@columbia.edu

Traders Joe is a technical analysis language that is used for modeling various algorithms.

Table of Contents

Introduction

Traders Joe is a language that provides high level technical analysis capabilities for any product like forex, security etc. This language is easy to use and has built in functions that help in technical analysis. Traders Joe understands mathematical notations and uses algorithms for modeling and simulating any trading models.

Traders Joe is written using OCaml and is optimized for technical analysis. It uses the state-of-the-art algorithms that are widely used by the technical analysis community.

Lexical Conventions

Character Set

Traders Joe uses ASCII character.

Keywords

Following are the keywords used by this language. These reserved words cannot be used as identifiers in the program. Keywords are case sensitive.

break	High	sma
candle	is_equal_to	sort_asc
Case	is_greater_than	sort_desc
Catch	is_less_than	sum
Close	is_less_than_or_equal_to	trace
Else	is_not_equal_to	try
elseif	Loop	while
ema	Low	alert
Fib	Median	
For	Open	

Constants

Integer Constants

An integer constant can contains series of number from 0 to 9. For example: 2000023

Floating Point Constants

Floating point are real numbers that contain fraction part. These numbers has an integer part, a decimal point and numbers that follow the decimal points. The character ‘e’ is optionally assigned after the decimal point.

String Constants

String constant are enclosed within double quotes. It can contain escape characters that is used for formatting messages.

Operators

Operators allow users to perform various operations. The following list contains various tokens that are used to assess relationship, perform logical operation, calculate numerical values or to assign values to a variable. Alternatively users can also use *is_equal_to, is_greater_than, is_less_than, is_less_than_or_equal_to, is_not_equal_to* to perform relationship operations.

Relational operators	
Equal To	==
Not Equal To	!=
Less Than	<
Greater Than	>
Less Than or Equal To	<=
Greater Than or Equal To	>=
Logical operators	
Not	!
And	&&
Mathematical operators	
Add	+
Subtract	-
Multiply	*
Divide	/
Power	^
Assignment operator	
Assign	=

Punctuators

Following punctuators are used to enclose or separate values or arguments. For example a string value is enclosed within double quotes “string value” and a character value is enclosed within single quotes ‘a’.

Punctuators	Description
" "	Encloses string value
' '	Encloses character value
( )	Groups arguments in a function
,	Separates arguments in a function

Comments

Comments are started with an open square bracket immediately followed by a start and terminated with a star immediately followed by a close square bracket. Any text between [* and *] are ignored. Comments helps programmers describe the function they are writing. Any code block enclosed between comments begin and end tag is also ignored.

For example:

[* *This comment is ignored by the program* ***]**

White Spaces

White spaces like tabs, line feeds etc are used to structure the code block. Developers may use them to align the code block to make their code readable.

Data Types

Simple Types

Type	Description
boolean	Boolean i.e. true or false
int	Integer values e.g. 1, 2, 3
float	Floating point values e.g. 1.2001
string	String value e.g. "string value"
char	Character value e.g. 'a'

Complex Types

Objects are classes that contain types, variables, functions etc that represents a model. Trading algorithms can be organized into classes and initialized as objects at runtime. This is useful when comparing the performance of two trading models.

Statements

Statements are sequence of statements or expressions that are executed sequentially. They can be complex or simple.

Compound Statements

Compound statement is collection of statements or expressions that are enclosed within a code block. A code block can be a function that is enclosed within a '*begin*' and '*end*' keyword.

Expression Statements

An expression is a statement that is separated by a operator. For example

```
var price == 1.2005
```

Control Statements

While loop

While loop allows conditional execution of the …

Example:

```
while close 0 is_less_than ema 20
begin
    [* statement *]
end
```

For loop

for *expression 1*; *expression 2; expression 3*
```
begin
    [* statement *]
end
```

If else condition

If… else are conditional statements and has the following syntax.

```
if $EURUSD close 0 is_less_than ema 10 0 then
        warn "Buy EURUSD"
else
        warn "No signal"
```

Functions

Built-in Functions

Alert

The alert function is used to return information to the user. A statement enclosed in double quotes should follow the alert command. Alert statement without proper string statement will return “Improper syntax error”.

Example:

```
alert “The opening price is greater than the previous closing price”
```

Trace

Trace is used to generate valuable information about the program and is used for debugging. Trace outputs suppressed when moving the code to the production environment. A string statement should follow the trace command.

Example:

```
trace “Reached the GetPrice function”
```

EMA

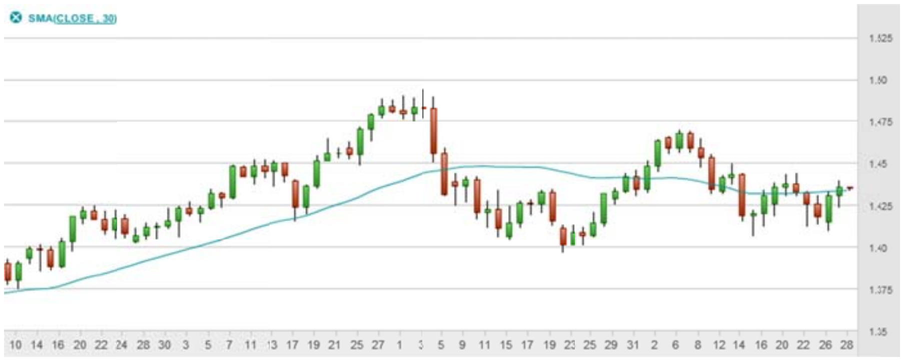
Exponential Moving Average (EMA) is a type of moving average that reacts faster to recent price changes. This function takes time period and calculating period as its parameter and returns a floating point number.

Example:

```
ema 10 1   [* returns the previous days EMA that is calculated using 10 days average *]
```

SMA

Simple Moving Average (SMA) is a type of moving average that reacts slower to recent price changes. This function takes time period and calculating period as its parameter and returns a floating point number.

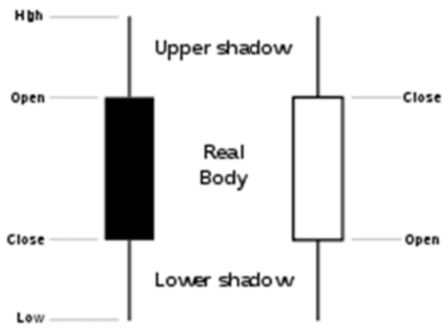


Example:

sma 10 1   [* returns the previous days SMA that is calculated using 10 days average *]

Candle

Candle sticks are patterns on the bar chart that describes the price movement. This function takes the calculating time and returns the type of candle stick patter found for that day. Dogi, Hammer are candle stick patterns.

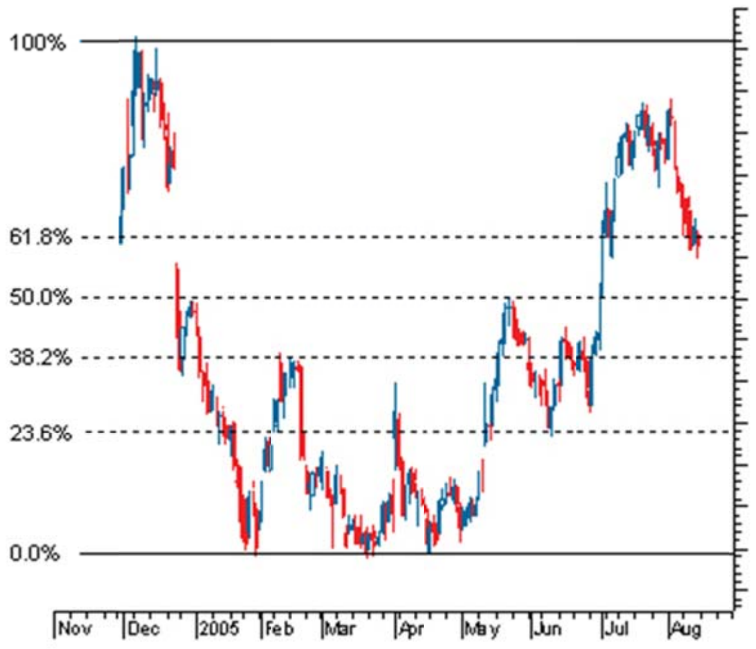


Example:

Candle 1   [* returns the previous candle pattern. *]

Fib

Fibonacci retracement is a very popular technical analysis tool and is based on key numbers identified by mathematician Leonardo Fibonacci in the thirteenth century. The function ‘Fib’ taken time period and a percentage as its parameter and returns a floating point number indicating the retracement level.



Example:

Fib 60 50%   [* returns the retracement at 50%. *]

Open

This function is used to find the opening price of a stock or currency pair. It takes time period integer as its parameter and returns an opening price floating point number.

Example:

Open 0 [* returns the current opening price. *]

Close

This function is used to find the closing price of a stock or currency pair. It takes time period integer as its parameter and returns a closing price floating point number.

Example:

Close 1 [* returns the closing price on the previous day. *]

High

This function is used to find the highest price of a stock or currency pair on a given day. It takes time period integer as its parameter and returns a floating point number.

Example:

High 1 [* returns the highest price on the previous day. *]

Low

This function is used to find the lowest price of a stock or currency pair. It takes time period integer as its parameter and returns a lowest price floating point number.

Example:

Low 1 [* returns the lowest price on the previous day. *]

Error Handling

Following are the built in errors that the system throws.

Error Code	Error Code	Description
1	FATAL_ERROR	Unexpected error occurred
2	SYNTAX_ERROR	Error in syntax
3	OUT_OF_BOUND	The data in the array you are referring does not exist
4	TYPE_MISMATCH	Data type is not in the correct format
5	MISSING_INPUT_FILE	Input file is missing
6	FORMAT_ERROR	Input file is not in the correct format

Errors can be handled using try and catch error block. The *try… catch* block without the *throw* block will suppress errors. It is a best practice to handle all errors appropriately. Following code catches error and suppresses it.

Suppress Error

```
[* Suppressing error *]
try
begin
        [* Statement *]
end
```

Catching Error

```
[* Suppressing OUT OF BOUND error *]
try
begin
        [* Statement *]
end
catch OUT_OF_BOUND
begin
        [* Handle error *]
end
```

Custom Error

```
[* Custom error *]
try
begin
        [* Statement *]
end
catch FORMAT_ERROR
begin
        throw “Error: Please check the format of the time series file.”
end
```

Time Series Inputs

Time Series data are inputs for various technical analysis operations. These inputs are stored in a file and passed to the language. This is a tab delimited file in the following format.

Symbol{tab}DataTime{tab}Open{tab}Close{tab}High{tab}Low

There is no restriction on the number of lines of data that the system can accept. Following is a sample time series data.

Symbol	DateTime	Open	Close	High	Low
EURUSD	1/5/2011	1.5095	1.5105	1.5109	1.5085
EURUSD	1/4/2011	1.5031	1.5095	1.5095	1.5021
EURUSD	1/3/2011	1.5085	1.5031	1.5085	1.5031
EURUSD	1/2/2011	1.5021	1.5085	1.5021	1.5031
EURUSD	1/1/2011	1.5001	1.5021	1.5021	1.5001

Usage:

```
$EURUSD = “C:\EURUSD.txt”
```

```
[* This will warn if the current ema 10 is less than todays close *]
if $EURUSD close 0 is_less_than ema 10 0 then
        warn "Buy EURUSD"
```

Scope Rules

Global variables:

The global variable many be declared anywhere in the program outside the local code blocks. These variables can be accessed by functions and will retain its value. A variable that is defined with the same name within a code block will however has its own value that is initialized inside a local code block.

Local variables:

A code block can be a function or a control statement like a while loop. Variables declared within these blocks are local to its container. If a code block is a control statement then it should begin with the '*begin*' keyword and end with a '*end*' keyword. Variable scope within a nested code block is local to its container.

Example:

```
var targetPrice = 1.2002                  [* Global variable *]
function GetTargetPrice
begin
        var targetPrice = 1.4000          [* Variable local to the GetTargetPrice function *]
        while close 0 is_lesser_than ema 20
        begin
                var targetPrice = 1.2500  [* Variable local to the control statement *]
        end
end
```

Appendix A

Scanner.mll

```
{ open Parser }

rule token =
parse [' ' '\t' '\r' '\n']                                    { token lexbuf }

| "[*"                                                        { comment lexbuf }
| "*]"                                                        { comment lexbuf }

| ';'                                                         { SEMICOLON }
| '('                                                         { LEFT_PAREN }
| ')'                                                         { RIGHT_PAREN }
| '['                                                         { LEFT_SQ_BRACKET }
| ']'                                                         { RIGHT_SQ_BRACKET }


| '+'                                                         { PLUS }
| '-'                                                         { MINUS }
| '*'                                                         { TIMES }
| '/'                                                         { DIVIDE }
| '^'                                                         { MATRIX_POWER }
| '.'                                                         { DECIMAL_POINT }

| '='                                                         { ASSIGNMENT }
| "=="                                                        { EQUAL_TO }
| "!="                                                        { NOT_EQUAL }
| "<"                                                         { LESS_THAN }
| ">"                                                         { GREATER_THAN }
| "<="                                                        { LESS_THAN_EQUAL_TO }

| "break"                                                     { BREAK_OUT }
| "case"                                                      { CASE_SWITCH }
| "else"                                                      { ELSE_CONDITION }
| "elseif"                                                    { ELSEIF_CONDITION }
| "end"                                                       { END_TERMINATE }
| "error"                                                     { ERROR_DISPLAY }
| "is"+ "_"+"less"+"_"+"than"                                 { IS_LESS_THAN }
| "is"+ "_"+"greater"+"_"+"than"                              { IS_GREATER_THAN }
| "is"+ "_"+"equal"+"_"+"than"                                { IS_EQUAL_TO }
| "is"+ "_"+"not"+"_"+"equal"+"_"+"to"                        { IS_NOT_EQUAL_TO }
| "is"+ "_"+"less"+"_"+"than"+"_"+"or"+"_"+"equal"+"_"+"to" { IS_LESS_THAN_OR_EQUAL_TO}
| "for"                                                       { FOR_LOOP }
| "while"                                    { WHILE_LOOP }
| "begin"                                    { BEGIN_LOOP }
| "end"                                      { END_LOOP }
| "if"                                       { IF_CONDITION }
| "and"                                      { AND_CONDITION }
| "then"                                     { THEN_CONDITION }
| "return"                                   { RETURN_STATEMENT }
| "switch"                                   { SWITCH_STATEMENT }
| "try"                                      { ERROR_TRY }
| "catch"                                    { ERROR_CATCH }
| "throw"                                    { ERROR_THROW }
| "trace"                                    { PRINT_TRACE }
| "warn"+' '+['A'-'Z']                       { WARNING }
| "print"+' '+['A'-'Z']                      { PRINT_MSG }
| "break"                                    {STATEMENT_TERMINATOR }

| "min"+" "+['0'-'9']                        { MINIMUM }
| "max"+" "+['0'-'9']                        { MAXIMUM }
| "sum"                                      { SUM_OF }
| "median"+" "+['0'-'9']                     { MEDIAN_PRICE }
| "sort_asc"                                 { SORT_ASC }
| "sort_desc"                                { SORT_DESC }

| "ema"+" "+['0'-'9']                        { EMA_AVERAGE }
| "sma"+" "+['0'-'9']                        { SMA_AVERAGE }
| "candle"+" "+['0'-'9']                     { CANDLE_TYPE }
| "fib"+" "+['0'-'9']                        { FIB_SEQ }
| "open"+" "+['0'-'9']                       { OPEN_PRICE }
| "close"+" "+['0'-'9']                      { CLOSE_PRICE }
| "high"+" "+['0'-'9']                        { HIGH_PRICE }
| "low"+" "+['0'-'9']                         { LOW_PRICE }

| '$'+['A'-'Z']                              { PRODUCT_SYMBOL }

| ['0'-'9']+ as lit                          { LITERAL(int_of_string lit) }
| eof                                        { EOF }
```