



DySen-Express

Language Reference

Document Version 1.8

www.fipertec.com

Contents

1	Introduction	4
2	Structure of an Express-Program.....	4
3	A Sentimentor Example.....	5
4	Adding an Express-Sentimentor to a Study.....	6
5	Working with the Express Editor	8
5.1.1	Keyboard Shortcuts	9
5.1.2	Verifying an Express Program	9
5.1.3	Understanding how Express Programs are Saved	9
6	Express Language Elements	9
6.1	Types	9
6.2	Reserved Words	10
6.3	Expressions	10
6.3.1	Numerical Expressions	11
6.3.2	Relational Expressions.....	11
6.3.3	Logical Expressions	11
6.3.4	String Expressions	11
6.4	Variable Declarations	12
6.5	Input Variables	12
6.6	Accessing Variables and Series data.....	13
6.7	Assignments	13
6.8	Assigning Sentiments.....	13
6.9	Predefined Series.....	13
6.10	Date and Time constants	14
6.11	Statements	14
6.12	Control Structures	14
6.12.1	if then	14
6.12.2	If then else	15
6.12.3	While Loop.....	15
6.12.4	For Loop	16
6.13	The Need for Speed.....	17
6.14	Interpretation – Computing the Sentiments.....	17
6.14.1	Interpretation Using the Built-in Schemes	17
6.14.2	Programming the Interpretation Explicitly	18

6.15	Plotting	19
7	A Blocker Example	20
8	A Stop Example	21
9	Built-In Functions and Procedures	22

1 Introduction

DySen-Express allows to program sentimentors that can be used in exactly the same way as the built-in sentimentors, i.e., they can be combined with other sentimentors and of course they can be optimized. Thus, the *Dynamite Sentimentor* framework in conjunction with the *Express* environment gives you an unparalleled power for specifying, optimizing, backtesting, and applying your trading ideas.

The scope of this document is to provide a description of the language *DySen-Express*. It is not intended to give an introduction into the theory and practice of programming or algorithms in general. A reader unfamiliar with the concepts of programming may have a look at introductory books for, e.g., Visual Basic, Pascal, Excel programming, or EasyLanguage for TradeStation. Once the main concepts like *variables*, *loops*, or *conditional expressions* are understood, working with *Express* will be very easy.

Besides a working knowledge of programming languages it is assumed that the reader is familiar with the terminology used in the “*DySen User’s Manual*”.

Users having experience with other programming languages for building indicators of trading systems should be aware of the overall “*Sentimentor*” approach used by *DySen* as this carries over to sentimentors programmed in *Express*, i.e., you will not find statements like “buy at open” – instead, an *Express* based *Sentimentor* is a building block used generating sentiments that eventually lead to trading actions through the combination with the *MetaSentimentor* and the applied trading approach.

2 Structure of an Express-Program

The way a *Sentimentor* is computed is as follows:

- Declare and initialize variables needed for the computation
- For each bar, carry out the required calculations.
- Compute the sentiments, i.e., how is the result of the calculation to be interpreted
- Define one or more charts to be plotted

An *Express* program reflects this by enforcing the following structure:

Variable Declarations Section

Calculation Section

Interpretation Section

Plot Section

3 A Sentimentor Example

Let's have a look at a simple Sentimentor that computes an exponential moving average (EMA). Buy and Sell sentiments are generated if the EMA crosses the close price. This Sentimentor is part of the DySen distribution.

<code>//(c) Fipertec</code>	1.
<code>Express EMA</code>	2.
<code>Vars</code>	3.
<code>input \$span (1, 200, 10);</code>	4.
<code>numeric factor (0);</code>	5.
<code>series ema;</code>	6.
<code>Calculation</code>	7.
<code>factor = 2 / (\$span + 1);</code>	8.
<code>if close[1] = void then //we need one lookback entry</code>	9.
<code> ema = close;</code>	10.
<code>else</code>	
<code> ema = factor * close + (1 - factor) * ema[1];</code>	11.
<code>interpretation TriggerLine(close, ema);</code>	12.
<code>plot (ema, blue, 2);</code>	13.
<code>plot (close, black, 1);</code>	14.

Explanation:

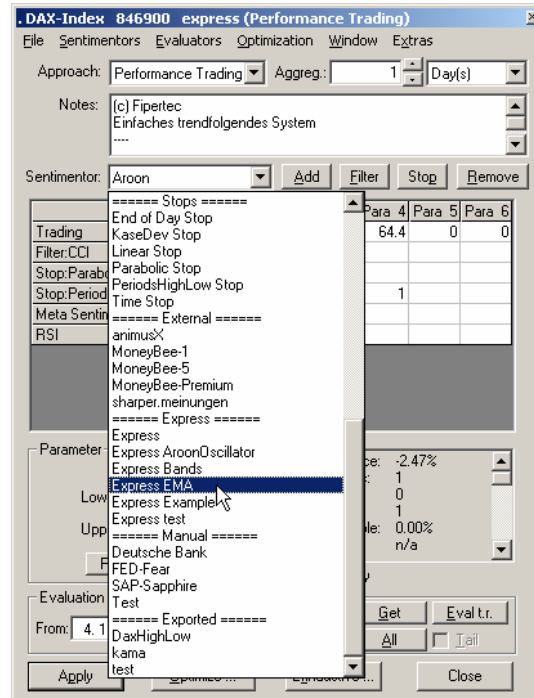
1. A double slash is used to start a comment reaching until the end of the line. It is also possible to use curly braces { } for comments anywhere in the code.
2. The program always starts with `Express <name>`, where `name` becomes the name of the Sentimentor used in the Dynamite-Dialog.
3. With the keyword `Vars` the Variable Declarations Section is started.
4. This line declares an integer variable that is subject to optimization. The minimal value is 1, the maximal value is 200, and the initial value is 10. The name of the variable, `span`, is displayed in the Dynamite-dialog. By using the `$` character, it is immediately visible in the code where optimization variables are used.
5. `factor` is defined as a numeric variable that may hold integer or float values. By using `(0)` it is initialized to zero. However, as Express guarantees to initialize numeric values to zero, the `(0)` could be omitted. Express is not case sensitive, i.e. `factor`, `Factor`, `FACTOR` all refer to the same variable. The same holds for keywords.
6. `ema` is defined as a `series` of float data. The series has the same length as the analyzed MasterChart. The elements of the series are

automatically initialized to zero. By using the `(val)` mechanism, `val` would be used as the initial value of all the elements of the series.

7. After the reserved word `Calculation` the statements for performing the computations begin. These statements are executed for each bar in turn, starting with the oldest or “left most”.
8. The `factor` for the EMA computation is determined, based on the input variable `$span`. The usual operators (`+`, `-`, `*`, `/`) and parenthesis can be used for mathematical operations.
9. Express defines a number of series implicitly, e.g., `close`, `open`, `high`, `low`, `volume`, to access the data of the MasterChart
For computing the EMA value of the actual period the EMA value of the previous period is also required. To access previous data of a series, an indexing is used: `close[1]` denotes the close of 1 one period ago. Generally, `close[n]` denotes the closing price of `n` periods ago, and `close` is simply a synonym for `close[0]`.
Assume we are calculating the EMA for the very first period then `close[1]` will not be available. In this case, the value of `close[1]` will be `void`, a reserved word that is used to identify non existing data.
The conditional statement `if` executes the then-part if the condition is fulfilled. In case the then-part consists of more than one statement, the then-part has to be started with `begin` and ended with `end`.
10. The first `ema` series element is assigned an initial value.
11. The `ema` for the current period is computed.
12. The conversion of the `ema/close` crossings into sentiments– called the *interpretation* – can be carried out by the standard interpreter *TriggerLine*. It would also be possible to compute the sentiments explicitly by assigning sentiment values to the predefined series `sentiment`.
13. The `ema` series is to be plotted in blue using a pen width of 2.
14. The `close` series is to be plotted in black using a pen width of 1.

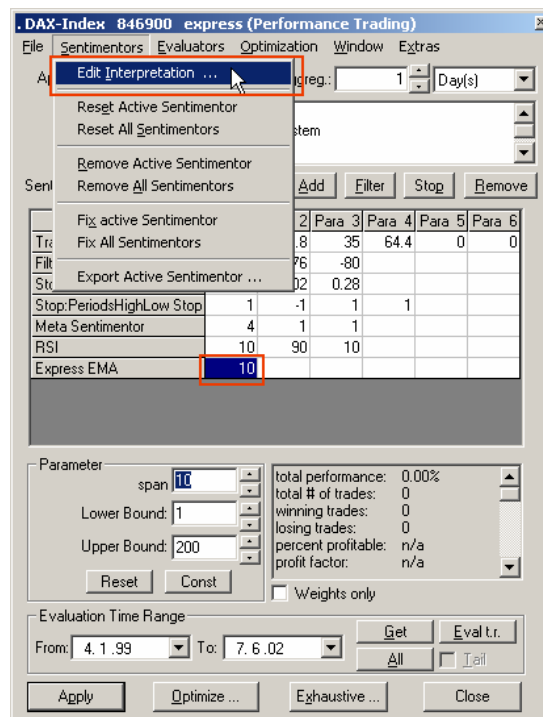
4 Adding an Express-Sentimentor to a Study

To add an Express Sentimentor to a study, choose the desired Sentimentor from the Sentimentor-list box and click on the Add or As Filter button.

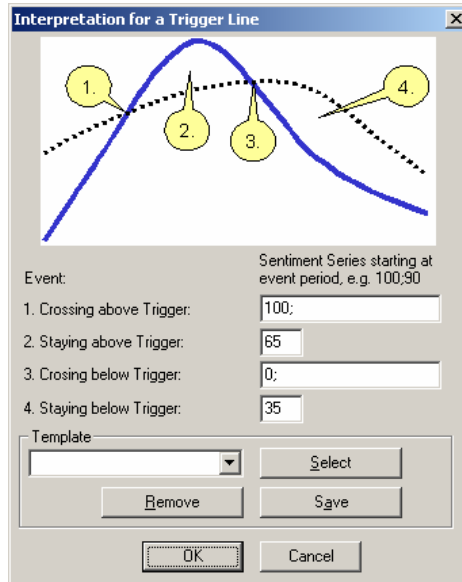


To edit the code of an Express Sentimentor double click the corresponding line in the Dynamite-tableau.

In case the Express Sentimentor applies a default interpretation scheme (see below), the scheme can be configured by first selecting the Express Sentimentor in the Dynamite-tableau and then choosing Sentimentor|Edit interpretation from the menu of the Dynamite-dialog:

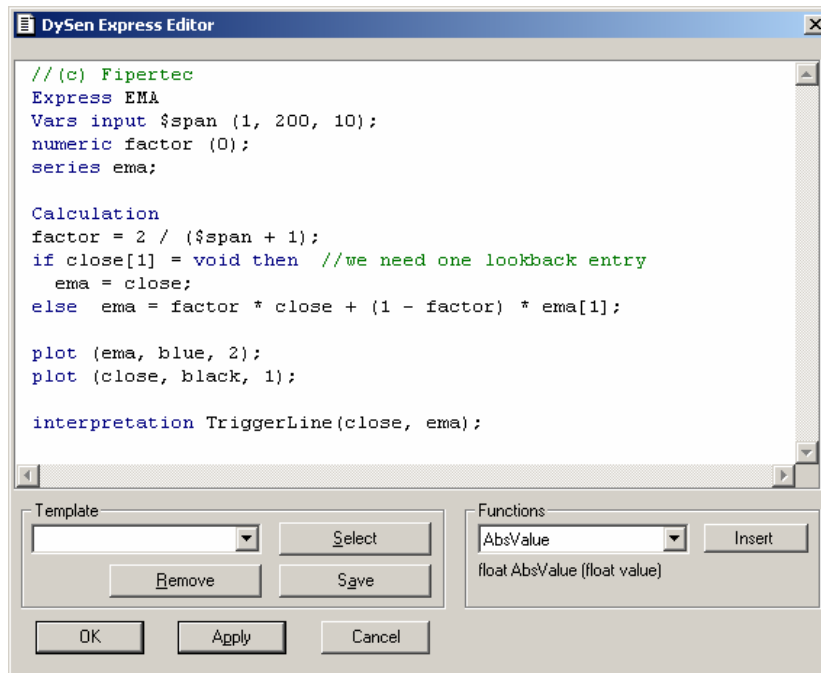


This will bring up the associated sentiment editor:



5 Working with the Express Editor

Double clicking on an Express Sentimentor in the Dynamite-dialog will open the Express editor:



The editor displays the reserved words of Express in blue and comments in green.

5.1.1 Keyboard Shortcuts

The default Windows shortcuts for cutting and pasting text can be applied, i.e., Ctrl-C to copy selected text, Ctrl-V to paste text, Ctrl-A to select the complete text, and Ctrl-S to save the text.

The Express editor also provides unlimited undo/redo using the standard Windows shortcuts Ctrl-Z for undo and Ctrl-Y for redo.

5.1.2 Verifying an Express Program

Clicking the Apply-button will execute the Express program and the results in terms of calculated series, plots, and signals are immediately displayed in the charts. In case the code contains errors, DySen will display appropriate error messages.

5.1.3 Understanding how Express Programs are Saved

When quitting the Express editor by clicking the OK-Button, the program is associated with the Express Sentimentor that has been double clicked to start the editor. Note that you need to save the study explicitly using File|Save from the Dynamite-dialog, otherwise the program might be lost.

However, the usual way is to save the Express program as a template before quitting the Express editor. This will save the programs text in the directory „Express“ below the installation directory of *Dynamite Sentimentor*. This will also make the program available as a Sentimentor in the Sentimentor-selection box of the Dynamite-dialog.

From within the Editor, you may load any template and adapt it to your specific needs.

It is good style to use the name of the Express program when saving it as a template. Suppose the program starts with

```
Express UltimateSenti
```

then *UltimateSenti* is the name of the Sentimentor. This name appears in the Dynamite-dialog and in the legend of its chart window. When saving the program as a template, it is desirable to use *UltimateSenti* as the templates name, although this is not enforced by the editor.

6 Express Language Elements

6.1 Types

Express is a so-called *typed language*, i.e. every *entity* representing a value is of a certain type. This allows *DySen* to catch a broad range of potential programming errors immediately.

The following types are supported by Express:

- `numeric`
A numeric value can be a float (e.g. 3.75) or an integer (e.g. 10). If needed, *DySen* automatically converts floats into integer values by stripping the fractional portion.
- `series`
A `series` is a series of elements of type `numeric`. Suppose the analyzed MasterChart consists of 200 bars then a series will also automatically consist of 200 elements. If the MasterChart is connected to a realtime data feed, then a `series` will grow automatically.
- `string`
A string a sequence of characters, like `Hello World!`. When using string constants in Express, the characters have to be enclosed in quotation marks: `"Hello World!"`
- `bool`
A boolean entity can take the values `true` or `false`.
- `time`
Entities of type `time` are used to work with times and dates, e.g.,

```
if timeOpen < 10:00 then sentiment = 50;  
if date = 2_8_2002 then sentiment = 100;
```

6.2 Reserved Words

Express uses a number of words that have a specific meaning. These words cannot be used for naming variables.

Currently the reserved words are:

```
express, sentimentor, blocker, stop, vars,  
calculation, interpretation, numeric, bool, senti_block,  
senti_flat, senti_pass, series, string, if, then, else,  
begin, end, for, to, downto, while, void, and, or, plot,  
plotline, plotcandles, plotbars, plothistogram
```

The following reserved words are for future use:

```
function, procedure, import, plothistogram
```

6.3 Expressions

An expression is a combination of *operators* and *operands*, like

```
5 + 3
```

where 5 and 3 are the operands and + is the operator.

Moreover, the value of a variable, a series, or the return value of a function are also expressions.

Express distinguishes between three types of expressions: numerical, string, and boolean.

6.3.1 Numerical Expressions

Numeric entities can be combined using the operators +, -, *, / with their usual mathematical meaning, i.e.,

```
5 * close - 3*close[1];
```

Parenthesis can be used to group expressions as in

```
(high + low + close) / 3
```

6.3.2 Relational Expressions

A relational expression evaluates to `true` or `false`, as in `close > open`.

The available relational operators are

Operator	Meaning
>	greater than
<	less than
=	equal to
>=	greater equal than
<=	less equal than
<>	not equal to

Relational operations may be used with numeric, time, and string entities. In the latter case the relational operation is evaluated with respect to the *lexicographical ordering*, i.e. "abc" is less than "xyz".

6.3.3 Logical Expressions

A logical expression is a combination of expressions evaluating to `true` or `false` with the operators `and` and `or`, e.g.,

```
(close > open) and (volume <= 1000)
```

```
(close > close[1]) and ((open > close) or (volume > 1000))
```

Note: Always use parenthesis to indicate exactly the grouping of the expressions. This enhances the readability of the code and avoids unnecessary programming errors.

6.3.4 String Expressions

The only operator for two entities of type string is + that is used for concatenating the strings, e.g.,

```
"Hello "+ "World!"
```

results in the new string "Hello World!".

6.4 Variable Declarations

All variables used by an Express program have to be declared in the `Variable Declarations` Section by stating the type and the name of the variable.

The name must begin with a character, e.g.,
`numeric weight;`

The initial value of a variable may be given in its declaration:

```
numeric weight(0.5);
```

To specify some variables of the same type the following notation can be used:

```
numeric weight, factor, delta;
```

Valid variable types are `numeric`, `bool`, `series`, and `string`. In case the initial values are not specified in the declaration, Express uses the following values:

Type	Initialized to
<code>numeric</code>	0
<code>bool</code>	<code>false</code>
<code>series</code>	all elements set to 0
<code>string</code>	<code>""</code> , i.e., the empty string

6.5 Input Variables

Input variables are numeric variables that are exposed to the outer world. They appear as parameters in the Dynamite-dialog and can be altered by the end user, just as any parameter of the built-in sentimentors. Moreover, the input variables are subject to optimization, so they need a minimal, a maximal, and an initial value. The declaration of an input variable is as follows:

```
input $<var name> (<min value>, <max value>, <initial value>);
```

Example:

```
input $span (1, 200, 20);
```

The `$` sign is used as prefix to indicate anywhere in the code that the referenced value is an input variable that is subject to optimization and hence is of great importance for the overall computation.

Very often, the input variables will only take integer values. However, sometimes you may need float values as input variables. For defining input variables of type float two more specifications are required: the precision and the step size. The latter is used by the optimization as the minimal change of the variables value. The declaration of an input variable holding float values is:

```
input $<var name> (<min value>, <max value>, <initial value>, <step size>, <precision>);
```

Example:

```
input $factor (1.00, 3.00, 2.00, 0.01, 2);
```

6.6 Accessing Variables and Series data

The value of a variable is accessed by simply using the variables name, as in

```
factor * delta
```

where `factor` and `delta` are declared as numeric variables.

An element of a variable of type `series` is always referenced *relative* to the currently processed bar. The syntax is as follows:

```
<series name> [n]
```

where n denotes a positive integer.

Example:

```
close[1] or close[span], where span is a numeric variable.
```

The example references the close price of the previous bar (1 bar ago).

To access the current bar, it is possible to use the abbreviation `close` instead of `close[0]`.

6.7 Assignments

A declared variable can be assigned a value using the `=` operator:

```
span = high - low;
```

When assigning a value to a variable of type `series`, Express automatically uses the element currently processed in the calculation section: Suppose `median` is a variable of type `series` and Express performs the calculation for the 25th bar of the MasterChart, then

```
median = (high + low + close) / 3;
```

assigns the result of the expression to the 25th element of the `median` series.

6.8 Assigning Sentiments

The sentiments of a Sentimentor programmed in Express have to be stored in the predefined series `sentiment`. As a sentiment has to be a value between 0 and 100, Express automatically enforces this, i.e., if a value greater than 100 is assigned, Express uses 100 instead, and in case a negative value is assigned, Express uses 0.

When computing sentiments, it is sometimes very convenient to assign a sentiment not only for the current period but also for the next 3, say, periods. This can be achieved using the following notation:

```
sentiment = [100; 90; 80;];
```

Note that this so-called *list assignment* is only valid for the predefined series `sentiment`.

6.9 Predefined Series

The following series referring to the MasterChart are always available:

open, close, high, low, volume. You may also use the abbreviations o, c, h, l, v.

The following series of type time are also available:

Series	Meaning
date	the date of the end of the period
dateOpen	the date of the beginning of the period
time	the time of the end of the period
timeOpen	the time of the beginning of the period

Finally, the series `sentiment` has to be used for storing the computed sentiments unless one of the default interpretation schemes is used.

With the exception of the sentiment series, all predefined series are *read only*, i.e., it is not possible to assign a value to them.

6.10 Date and Time constants

Time constants may be used in boolean expressions. A time has the format HH:MM or HH:MM:SS

Example:

```
14:15    14:50:40
```

If the seconds are omitted they are automatically set to 0.

The format of a date is:

```
DD_MM_YYYY
```

Example:

```
22_10_2002
```

Time and date constants can be used in boolean expressions, as in

```
if time < 10:00 then
    sentiment = 50;           //don't buy during the first hour of the session
```

6.11 Statements

A statement is a complete Express instruction composed out of reserved words, operators, operands and ended by a semicolon. E.g.

```
delta = high - low;
```

6.12 Control Structures

6.12.1 if then

The if control structure is used to execute statements only if a specified condition is met. The syntax is:

```
if <boolean expression> then
    <statement>;
```

If several statements are to be executed the following syntax has to be used:

```
if <boolean expression> then
begin
    <statement>;
    <statement>;
    ...
    <statement>;
end
```

Example:

```
if close > open then
    upMoves = upMoves + 1;
```

6.12.2 If then else

The if control structure may also contain statements to be executed in case the condition is *not* met:

```
if <boolean expression> then
    <statement>;
else
    <statement>;
```

Again, use `begin` and `end` to group a number of statements to be executed.

Example:

```
if close > open then
    upMoves = upMoves + 1;
else
begin
    downMoves = downMoves + 1;
    downVol = downVol + volume;
end;
```

6.12.3 While Loop

The syntax of the while loop is as follows:

```
while <boolean expression>
begin
    <statement>;
    <statement>;
    ...
    <statement>;
end
```

The statements are executed until the <boolean expression> evaluates to true.

Example:

```
lastHigh = 1;
vol = 0;
while (close[lastHigh] <> void) and (close > close[lastHigh])
begin
    vol = vol + volume[lastHigh];
    lastHigh = lastHigh + 1;
```

end

Note: Double check that the <boolean expression> will finally evaluate to `false`, otherwise the while loop would run endlessly and the system will be blocked. There is no way for DySen to verify the finiteness of a <boolean expression>. Therefore, if a While loop does not terminate within five seconds, the Express program is terminated by DySen.

6.12.4 For Loop

The syntax of a `FOR` loop is as follows:

```
for <variable> = <start value> to <numerical expression>
begin
  <statement>;
  <statement>;
  ...
  <statement>;
end
```

At the beginning of the `FOR` loop, <variable> is set to the <start value>. If <variable> does not exceed <numerical expression> then the statements are executed and <variable> is increased by one. This process repeats until <variable> finally exceeds <numerical expression>.

Example:

```
upMoves = 0;
for i = 0 to 9
begin
  if close[i] > open[i] then
    upMoves = upMoves + 1;
end
```

Sometimes it is desirable to *decrease* the <variable> and to stop the loop if the <variable> falls below <numerical expression>. This can be achieved by using the following variant of the For loop:

```
for <variable> = <start value> downto <numerical expression>
begin
  <statement>;
  <statement>;
  ...
  <statement>;
end
```

Note: Double check that the <variable> will finally exceed <numerical expression> (or falls below it in case of the `downto` variant). There is no way for DySen to verify the finiteness of a For loop. Therefore, if a For loop does not terminate within five seconds, the Express program is terminated by DySen.

6.13 The Need for Speed

Whenever `While` or `For` loops are used, make sure that the computation you are carrying out is *efficient*. It is very easy to implement a calculation in a naive way that works, but that requires an enormous amount of computation time.

Take for example the calculation of a 50-bar moving average. The naive approach would sum up the close price of the current and the previous 49 bars and then divide the result by 50. A more intelligent approach would take advantage of the fact that whenever moving to the next bar, the new sum could be computed by subtracting the “leftmost” price and adding the price of the current bar.

Hence, the naive approach is 50 times slower (*in words: fifty*) than the more intelligent approach. For a 200-bar moving average it would be 200 times slower. Now suppose what happens if you use the “naive” implementation within an optimization...

Quite often some calculations can be performed after all bars have been processed, e.g., you want to apply a moving average on a complete series you have computed. This can be achieved easily by using the boolean built-in function `IsFinalBar()`:

```
...
series result;
input $span (1, 200, 10);
Calculation:
..result = ...;
  if IsFinalBar() then //true, if currently the final bar is processed
    MovingAverage (result, result, $span); //built-in function
```

6.14 Interpretation – Computing the Sentiments

The main aspect of a `Sentimentor` is obviously the computation of a sentiment for each period. This is done in the `Interpretation` Section of an `Express` program. The `Interpretation` Section starts is introduced with the reserved word `Interpretation`.

6.14.1 Interpretation Using the Built-in Schemes

Very often the computation of the sentiments can be performed by one of the built-in interpretation schemes that are also used by the built-in sentimentors. Moreover, when using a built-in scheme, the corresponding editor for configuring the scheme details is available. So relying on a built-in scheme greatly simplifies the programming of an `Express` `Sentimentor`.

A built-in scheme can be called like a normal function.

Example:

```
interpretation TwoThresholds (mySeries, $upZone, $downZone);
Or
interpretation TriggerLine (close, mySeries);
```

The `TriggerLine` scheme would compute the sentiments based on the close series crossing the series `mySeries`.

In case a built-in scheme requires input variables they have to be provided as parameters.

The following built-in schemes are available:

Built-in scheme	Typical usage
<code>TwoThresholds (series curve, input upThreshold, input downThreshold)</code>	RSI
<code>TriggerLine (series curve, series trigger)</code>	Crossing MA
<code>Swing (series curve, input spanLeft, input spanRight)</code>	Momentum
<code>Bands (series curve, series lower, series upper)</code>	Bollinger Bands

When using a built-in scheme, the plot statements may be omitted – Express will automatically plot the series used in the built-in scheme. However, if at least one plot statement is given, this standard mechanism is not applied.

6.14.2 Programming the Interpretation Explicitly

In case no built-in scheme matches the intended interpretation the sentiments can be computed explicitly using the syntax:

```
interpretation
begin
  <statement>
...
  <statement>
end
```

Note the `begin` and `end` surrounding the statements for computing the sentiments.

The process for computing the sentiments equals the process in the calculation section, i.e., the statements are executed for each bar, starting with the oldest („left most“) bar.

The sentiments have to be assigned to the predefined series `sentiment`. DySen initializes the elements of the sentiment series with 50, i.e., *neutral*.

Example

```
interpretation
begin
  if CrossesAbove (close, mySeries) then
    sentiment = 100;
end
```

Instead of assigning the sentiment for the current bar only, it is also possible to assign sentiments for the following bars. With this technique, an *event* can be of

significance not only in the period where it happens but also in the following periods.

Example:

```
interpretation
begin
  if CrossesAbove (close, mySeries) then
    sentiment = [100; 90; 80;];
  else if close > mySeries then //staying above mySeries
    if sentiment = 50 then //do not overwrite crossing event
      sentiment = 65;
    end
  end
end
```

The so-called *list assignment*

```
sentiment = [value; value;...;];
```

is only valid for the series `sentiment`.

6.15 Plotting

The final statements of an Express program are one or more `plot` statements following the syntax:

```
plot (<series name>, <color>, <pen width>);
or
plotline (<constant or variable>, <color>, <pen width>);
```

Example:

```
plot (mySeries, blue, 2);
plotline ($threshold, green 1);
```

The following colors are predefined: `red`, `green`, `black`, `blue`. A color can also be defined as a so-called RGB-value (RGB= Red-Green-Blue) using the syntax:

```
plot (<series name>, <red>, <green>, <blue>, <pen width>);
```

The red/green/blue values are integers in the range of 0 to 255 defining the strength of the respective color.

Sometimes it is interesting to plot candles or bars based on real or modified price data. This can be achieved by the following plot routines:

```
plotcandles (<open series>, <close series>, <high series>, <low series>);
or
plotbars (<open series>, <close series>, <high series>, <low series>);
```

7 A Blocker Example

In addition to the sentiment values in the range from 0 to 100, *Dynamite Sentimentor* supports two specific sentiment states that are used in conjunction with filters:

- **BLOCK**
Long *and* Short signals are rejected
- **FLAT**
Long *and* Short signals are rejected. In addition, a possible open position is closed.

When working with Manual Sentimentors *Dynamite Sentimentor* allows the usage of these states, e.g, to prohibit the trading at a certain daytime.

The programming of such a Blocker using Express is shown in the following example:

<pre>Express Blocker VolCheck Vars series twoPeriodVol; Calculation if CurrentBarIndex () > 1 then twoPeriodVol = vol[1] + vol; interpretation begin if twoPeriodVol < 10000 then sentiment = senti_block; else if twoPeriodVol > 5000000 then sentiment = senti_flat; //just for illustration else sentiment = senti_pass plot (twoPeriodVol, blue, 2);</pre>	<p>1.</p> <p>2.</p> <p>3.</p> <p>4.</p>
--	---

Explanation:

1. The keyword `Blocker` declares this sentimentor to work as a Blocker. This enables the usage of the sentiment constants `senti_block`, `senti_flat` and `senti_pass`. Moreover, this keyword ensures that the sentimentor can only be added as a Filter into a study.
2. The constant `senti_block` makes sure that Long and Short signals are rejected (blocked).
3. The constant `senti_flat` makes sure that Long and Short signals are rejected and a possible open position will be closed.
4. The constant `senti_pass` does not filter any signal.

8 A Stop Example

The implementation of a pricebased stop, i.e. a sentimentor that computes a stop price, is demonstrated with the following example:

Express Stop Simple	1.
vars	
input \$increase (1, 25, 10);	
series ma;	
calculation	
if IsFirstBar () then	
MovingAverage (close, ma, 10);	
if MarketPosition() = 1 then //long	2.
begin	
if IsIntradayEntry() then //we just opened the position	3.
SetStopPrice (EntryPrice() - 15);	4.
else	
SetStopPrice (ma - 100 + \$increase* BarsSinceEntry());	
end	
else if MarketPosition() = -1 then //short	
begin	
if IsIntradayEntry() then //we just opened the position	
SetStopPrice (EntryPrice() - 15);	
else	
SetStopPrice (ma + 100 - \$increase* BarsSinceEntry());	
end	
	5.

Explanation:

1. The keyword `Stop` declares this sentimentor to work as a price based stop. This enables the usage of functions only available for this kind of sentimentors. Moreover, this keyword ensures that the sentimentor can only be added as a `Stop` into a study.
2. The function `MarketPosition()` informs about the current position:
 - 1 = long
 - 0 = flat
 - 1 = short
3. The boolean function `IsIntradayEntry()` returns `true` in case the position has just been opened in the current, not yet closed period. Sometimes it is necessary to use a different scheme for calculating the stop price for the initial period, e.g., based only on the entry price. In case the position is not closed within this initial period, the stop price for the next period to come will be calculated again.
4. The function `SetStopPrice()` makes the computed stop price available to *Dynamite Sentimentor* that will choose the tightest stop among all used stops in the current study.
5. For `Stop` sentimentors there is no `Interpretation Section` and no `Plot Section`

9 Built-In Functions and Procedures

Express provides a number of built-in functions that can be called from within an Express program. If a built-in function requires parameters, DySen checks if the provided parameters match the *function definition*. A function definition declares how a function is to be called.

The function definition of the `Max()` -function is given as:

```
float Max (float value1, float value2)
```

Hence, the return value of the function `Max` is of type `float`. The function takes two parameters, both of type `float`. Recall that Express automatically converts integer values to float values if needed, so

```
Max (close, 5000)
```

is a valid call, as 5000 would automatically be converted into a `float` value.

Functions returning a value can be used in expressions, as in

```
mySeries = Max (open, close) * 2;
```

Functions that do not return a value are also called *procedures*. A procedure call cannot be used in an expression. Instead, it forms a complete statement:

```
MovingAverage (mySeries, mySeries, $span);
```

The definition of `MovingAverage` is

```
void MovingAverage (series source, series target, int span)
```

The return value `void` indicates that there is in fact no return value.

The names of the parameters in the definitions are chosen such that they indicate their role for the function – they have no other specific meaning.

Even if a function does not receive parameters, the parenthesis have to be used:

```
index = CurrentBarIndex();
```

The following built-in functions are available:

Definition: float **AbsValue** (float value)

Meaning: Returns the absolute value of `value`.

Example: `AbsValue (-3.7)` returns 3.7; `AbsValue (5)` returns 5

Definition: float **ArcTangent** (float value)

Meaning: Returns the arcus tangent of `value`.

Example: `ArcTangent (2.7475)` returns 70.

Definition: float **Atr** (int span)

Meaning: Returns the Average True Range of the MasterChart for the actual and previous `span` bars.

Definition: void **Bands** (series series, series lower, series upper)

Meaning: Standard interpretation scheme where the sentiments are computed based on an upper and lower series

Example: interpretation Bands (close, myLower, myUpper);

Definition: float **Ceiling** (float value)

Meaning: Returns the smallest integer greater than `value`.

Example: Ceiling (2.95) returns 3

Definition: float **Cosine** (float value)

Meaning: Returns the cosine of `value` degrees.

Example: Cosine (45) returns 0.7071

Definition: bool **CrossesAbove** (series curve, series trigger)

Meaning: Returns true (if curve[1] <= trigger [1]) and (curve > trigger), false otherwise

Example: if CrossesAbove (mySeries, close) then sentiment = 100;

Definition: bool **CrossesAboveThreshold** (series curve, float threshold)

Meaning: Returns true (if curve[1] <= threshold) and (curve > threshold), false otherwise

Example: if CrosseAboveThreshold (mySeries, 70) then sentiment = 100;

Definition: bool **CrossesBelow** (series curve, series trigger)

Meaning: Returns true (if curve[1] >= trigger[1]) and (curve < trigger), false otherwise

Example: if CrossesBelow (mySeries, close) then sentiment = 0;

Definition: bool **CrossesBelowThreshold** (series curve, float threshold)

Meaning: Returns true (if curve[1] >= threshold]) and (curve < trigger), false otherwise

Example: if CrossesBelowThreshold (mySeries, 30) then sentiment = 0;

Definition: int **CurrentBarIndex** ()

Meaning: The index of the currently processed bar. The first bar has the index 0.

Example: highDiff = CurrentBarIndex() – IndexOfHighest (close, 10);

Definition: float **EntryPrice** ()

Meaning: The entry price for the current position.

Example: SetStopPrice(EntryPrice() – 0.05);

Definition: int **DayOfWeek** (time time)

Meaning: The index of the day within 'time' in the week. Monday = 1, Tuesday= 2,

Example: if DayOfWeek(date) = 5 then sentiment = 50; //don't buy on Fridays.

Definition: float **Floor** (float value)

Meaning: Returns the largest integer smaller than 'value'.

Example: Floor (2.95) returns 2

Definition: float **Highest** (series series, int span)

Meaning: Returns the highest value in series for the elements series[0], ... series[span – 1]

Example: tenBarHigh = Highest (close, 10);

Definition: **IndexOfHighest** (series series, int span)

Meaning: Returns the index of the highest value for the elements series[0], ... series[span – 1]

Example: highDiff = CurrentBarIndex() – IndexOfHighest (close, 10);

Definition: **IndexOfLowest** (series series, int span)

Meaning: : Returns the index of the lowest value for the elements series[0], ... series[span – 1]

Example: lowDiff = CurrentBarIndex() – IndexOfLowest (close, 10);

Definition: bool **IsFinalBar**()

Meaning: Returns true if currently the final bar is processed.

Example: if IsFinalBar () then MovingAverage (mySeries, mySeries, \$span);

Definition: bool **IsFirstBar()**

Meaning: Returns true if currently the first bar is processed.

Example: if IsFirstBar () then MovingAverage (close, mySeries, \$span);

Definition: bool **IsIntradayEntry()**

Meaning: Returns `true` in case the position has just been opened in the current, not yet closed period. Sometimes it is necessary to use a different scheme for calculating the stop price for the initial period, e.g., based only on the entry price. In case the position is not closed within this initial period, the stop price for the next period to come will be calculated again.

Example: if IsIntradayEntry () then SetStopPrice(EntryPrice() – 0.05);

Definition: bool **IsNonZero(float value)**

Meaning: Returns true if `value` >= 0.001. Never test with „= 0“, because due to rounding errors this condition is very rarely met.

Example: if IsNonZero (a * b) then val = sum / (a * b);

Definition: bool **IsZero(float value)**

Meaning: Returns true if `value` < 0.001. Never test with „= 0“, because due to rounding errors this condition is very rarely met.

Example: if Not IsZero (a * b) then val = sum / (a * b);

Definition: float **Log** (float value)

Meaning: Returns the natural logarithm of `value` or void if `value` <= 0.

Example: Log (1000) returns 6.9078

Definition: float **Lowest** (series series, int span)

Meaning: Returns the lowest value in series for the elements series[0], ... series[span – 1]

Example: tenBarLow = Lowest (close, 10);

Definition: int **MarketPosition** ()

Meaning: Returns the direction of the current position:

1 = long

0 = flat

-1 = short

Example: if MarketPosition() = 1 then SetStopPrice (low – 0.01);

Definition: float **Max** (float value1, float value2)

Meaning: Returns the maximum value of `value1` and `value2`

Example: Max (3, 7.5) returns 7.5

Definition: float **Min** (float value1, float value2)

Meaning: : Returns the minimum value of `value1` and `value2`

Example: Min (3, 7.5) returns 3

Definition: **MovingAverage** (series source, series target, int span)

Meaning: Computes the \$span-bar MovingAverage of `source` and writes the result into `target`. This function should only be used in conjunction with IsFirstBar() or IsFinalBar().

Example:

if IsFirstBar () then MovingAverage (close, mySeries, \$span);

if IsFinalBar () then MovingAverage (mySeries, mySeries, \$span);

Definition: void **Plot** (series curve, color name, int penWidth)

void **Plot** (series curve, int red, int green, int blue, int penWidth)

Meaning: Plots the series `curve` using in the specified color and pen width.

Example: Plot (close, green, 2);

Plot (close, 128, 128, 128, 1); //grey

Definition: void **PlotLine** (float value, color name, int penWidth)

Meaning: Plots a horizontal line at level `value` using the specified color and pen width..

Example: PlotLine (\$threshold, red, 2);

PlotLine (100, 150, 0, 0, 1); //dark red

Definition: **PlotBars** (series open, series close, series high, series low)

Meaning: Plots a bar chart using the specified series.

Example: PlotBars (myOpen, myClose, high, low);

Definition: **PlotCandles** (series open, series close, series high, series low)

Definition: colo

Meaning: Plots a candle stick chart using the specified series.

Example: PlotCandles (myOpen, myClose, high, low);

Definition: float **Power** (float value, float exponent)

Meaning: Returns `value` raised to the power `exponent`.

Example: Power (3, 3) returns 27

Definition: float **PrevDayHigh/Low/Open/Close/Vol** ()

Meaning: Returns the High/Low/Open/Close/Vol of the previous day or `void` in case the data is not available

Example: `yesterdayMedian = (PrevDayHigh() + PrevDayLow() + PrevDayClose()) / 3`

Definition: void **SetStopPrice** (float value)

Meaning: Defines the stop price for the current period in case the position has just been entered or for the next period in case the position has been entered before this period.

Example: `SetStopPrice (low[-1]);`

Definition: void **ShowTip** (string message)

Meaning: Anchors `message` to the current bar. The message is displayed in a popup window when the cursor is above the bar. To indicate a new line use the character sequence `\n`.

Example: `if CrossesAbove (mySeries, 70) then ShowTip („Entered upper zone!\nWait for confirmation.“);`

Definition: float **Sign** (float value)

Meaning: Returns the sign of value.

Example: `Sign (-3) return -1; Sign (5) returns 1; Sign (0) returns 0`

Definition: float **Sine** (float value)

Meaning: Returns the sine of `value` degrees.

Example: `Sine (70) returns 0.9397`

Definition: float **SquareRoot** (float value)

Meaning: Returns the square root of value or `void` if `value` < 0.

Example: `SquareRoot (4) returns 2`

Definition: float **StdDev** (series source, series target, int span)

Meaning: Computes the standard deviation of the values source, source [1], ..., source[span-1] saves the result in target.

This function should only be used in conjunction with IsFirstBar().

Example: StdDev (close, myseries, 10);

Definition: float **Sum** (series series, int span)

Meaning: : Returns the sum of the elements series[0], ... series[span – 1]. Returns void in case one or more required elements are void.

Example: amount = Sum (mySeries, \$span);

Definition: void **Swing** (series series, input spanLeft, input spanRight)

Meaning: Standard interpretation scheme where the sentiments are computed based on swings in series

Example: interpretation Swings (mySeries);

Definition: float **Tangent** (float value)

Meaning: Returns the tangent of `value` degrees.

Example: Tangent (70) returns 2.7475.

Definition: **TriggerLine** (series curve, series trigger)

Meaning Standard interpretation scheme where the sentiments are computed based on the crossings of `curve` and `series`.

Example: interpretation TriggerLine (close, mySeries);

Definition: void **TwoThresholds** (series series, input upThreshold, input downThreshold)

Meaning: Standard interpretation scheme where the sentiments are computed based on zones defined by two thresholds.

Example: interpretation TwoThresholds (mySeries, \$upperZone, \$lowerZone);