ALICE REFERENCE MANUAL

Author: Tim Cooper
Reviewer: Thomas Jones, Andi Hakim , Audris Siow
Alice Version: 4.2
Date: 4 March, 2008

Document History

Version	Date	Summary of Changes
4.1	26.05.01	TCO added the 'uvolume' function and clarified the 'volume' function.
4.1	30.05.01	TCO merged the Derivatives query functions with the derivatives functions.
4.1	31.05.01	TCO clarified the 'dummy…' functions.
4.2	18.09.01	TCO clarified and reorganised some orderbook functions
4.2	19.10.01	TJO fixed definitions for indexval functions [require index argument]
4.2	22.10.01	AHA added definitions for findordervol, ask/bid(board), weighting_in_index, InControlGroup and many many other functions
4.2	13.02.02	TJO updated index page numbers, small mods, re-released
4.2	14.02.02	TJO added index entries for the newly added functions
4.2	05.03.02	ASI added the set distsetaccuracy(n) function
4.2	8.11.02	TCO added 'executetime'.
4.2	20.11.02	AWA added 'bidvolatdepth(n)', 'askvolatdepth(n)', 'bidvolatstep(n)' and 'askvolatstep(n)'.
4.2	13.12.02	CRI Fixed the note on array names
5.0	6.1.03	TCO documented '10 trdays'.
5.0	27.01.03	LCH documented 'infotranstype'
5.0	3.02.03	LCH added note on 'printcsv': the effect of commas in "Long Names"
5.0	5.02.03	LCH added 'exp(X)' to mathematical functions.
5.0	13.05.03	LCH removed unused command-line parameters
5.0	23.05.03	LCH undid last mod and added a column for Unix/Win or Unix commandline parameters.
5.0	3.06.03	TCO undocumented value(number) et al, and documented value(date1,date2) et al. The 'number' versions of these functions are obsolete and will be removed sometime.
5.4	18.06.2003	TCO documented the new 'before entord/before trade/before etc.' when-clauses.
5.4	25.06.2003	LCH documented the right click on include file feature.
5.4	2.07.2003	TCO clarified the 'issuedate' function.
5.4	11.7.2003	TCO documented the new options 'greeks' functions, and the capm_beta/capm_alpha functions, and clarified that 'board' does not return simply "first 4 letters of board name".
5.4	4.8.2003	CCO additions to TCO's "on amdtrade"
5.4	14.8.2003	TCO documented the 'debug' command and 'childmkt' function.
5.4	29.9.2003	TCO added the "on pendingord" when-clause.
5.4	18.12.2003	TCO added the "herald" functions.
5.4	24.5.2004	SWU added the description for closing file functionality in printto and printcsv
5.15	8.6.2004	CCO added note for blackscholes, at end vs. at dayend
5.18	19.7.2004	TCO renamed bestbidorderid → PriorityBid.
5.19	24.09.2004	LCH added tagfiled client functions
5.20	22.02.2004	TCO clarified the measurevolatily functions.
5.21	24.4.2006	TCO added "on refupdate" (and removed "on settlement").
5.25	6.10.2006	LCH added some code examples
5.26	27.11.2006	MPR added official functions

Table of Contents

1. SUMMARY

This Alice manual is suitable for ALICE in the SMARTS v5.0+ release.
Alice is a programming language designed specifically for
(a) creating alerting rules based on historical or realtime trading data
(b) generating reports based on historical or realtime trading data.

Alerts generated by **SMARTS** are specified using *Alice*. This gives many advantages over alerts coded in a normal compiled language such as C.
Some advantages of *Alice* rather than C.

- It is the world's first language which specialises in the creation of alerts.
- It has a type-system tailored for financial concepts such as ***price***, ***volume.***
- The alerts may be easily customised by the user.
- Bugs cannot be introduced into the rest of **SMARTS** by changing the alerts.
- The Intellectual Property of the rules is kept in one place - the *Alice* source. This is separated from the rest of **SMARTS**.
- Rules can be quickly created and tested without the need for compilation.

Stylistic Conventions used in this document:

- *Alice* keywords and functions are in **Arial Bold**.
- *Alice* program contents are in Arial.
- **SMARTS** applications are in **BOLD ALL CAPS TIMES NEW ROMAN**.
- Examples are in bold and highlighted in **black** is the function being described;

Naming conventions used in this document:

- "security" is equivalent to "stock" or "instrument" or "issue", ie. any financial instrument.
- "trade" is equivalent to "deal", ie. a transfer of some quantity of securities for money.
- "house" is equivalent to "firm", "broker", "bank" etc., ie. the company which is directly involved in trading.
- "trader" is equivalent to "user", "operator" or "dealer", ie. the individual person responsible for entering orders and performing trading.

The source for the alerts is contained within the *Alice* file. **SMARTS** provides a full set of pre-written alert rules which can be used by the surveillance department initially, until they become more familiar with the *Alice Language* and their market place. Later, these rules may be modified or new ones may be created in **ALDIT** (the Alert Editor).

2. STRUCTURE OF AN ALERT RULE

An *Alice* program consists of a sequence of alert rules. A rule is a *when clause* followed by one or more *statements*. Generally speaking, these alert rules can come in any order.
Following is an example of an alert rule to detect 'ramping' behaviour for a market which closes at 1pm.

```
on trade
    if time > 12:45 then
        if price > 1.10 * price(12:45) and
          price - price(12:45) > $0.50 and
          flags(-OF-od) then
            alert 108, “RAMPING”, “LOW PRICE RAMPING:
                price change [change(price, price(12:45))] on
                [price(12:45)] within last [ceiling(float(time-12:45)/60)]
                minutes (from 12:45 to [time])”
        end if
    end if
end on
```

2.1 The when clause

For each alert, the first component to be defined is the *when clause*. This determines when the test condition ought to be evaluated. A complete list of *when clauses* is defined in section 4.
In the example above, the *when clause* in the alert rule is **on trade**, which means that the test condition is checked after every trade. This is the most common time for a rule to be evaluated. Alternative *when clauses* include **on order**, **every** 5mins, **at** 12:45.
Each of the *when clauses* must be concluded at the end of the statement. This is done by typing **end on**, **end every**, **end at** etc.

2.2 Statements

A *statement* determines the conditions to be evaluated. There are many types of *statements*, for example **if** *statements*, **alert** *statements* and **print** *statements*. A single alert rule can contain multiple *statements*. Each of these is described further in section 5.
Generally an alert rule contains an **if** *statement* and an **alert** *statement*.

2.2.1 The if statement

In the above example, there are two **if** statements.
The first is used to evaluate whether the time is past 12:45pm. If the time is indeed past 12:45pm, the statement following the keyword **then** is executed.
The second **if** statement is used to evaluate the price at which the trade was executed. More specifically, it says (in human speak):

"If the price of the trade is at least 10% higher than the price of the same security at 12:45, and
the price rise is at least 50 cents, and
the trade is neither an off-market trade or an odd-lot trade then"

If the conditions specified in the **if** *statements* have been satisfied, the alert rule will then proceed to the **alert** *statement*.
Similar to the *when clause*, **if** *statements* must also be concluded by typing **end if**.

2.2.2 The alert statement

The last *statement* in an alert rule is generally the **alert** *statement*. If all conditions leading up to the **alert** *statement* is satisfied, an alert is sent to the alerting system and will appear in **ALMAS** (the Alert Management System). The various parameters of the alert are determined by the expressions contained in the **alert** *statement*.
In the example, the **alert** *statement* defines the

alert code	108
short name	RAMPING
full details	LOW PRICE RAMPING: price change [*w*] on [x] within last [y] minutes (from 12:45 to [*z*])

Expressions between [and] symbols in the **alert** *statement* represents expressions which will be replaced by numbers or values calculated based on the current trading condition.
For more details about the **alert** *statement* refer to section 5 and 0.

2.3 Conclusion

Above you have seen an example of a simple alert rule. With *Alice* it is possible to express all the behaviours which you want the alerting system to monitor, and access benchmarks to establish a concept of what is normal, and in turn what is unusual. The rest of this document provides a complete list of all the functions, statements, operators etc. which exist in *Alice Language*. They serve as a reference manual for the *Alice* programmer.

3. OTHER COMPONENTS OF AN ALICE PROGRAM

3.1 User Parameters

At the top of the *Alice* program is an optional section, know as the *user parameter* section. This section stores all information which 'parameterises' an *Alice* program.
The *user parameter* section begins with the keyword **userparams**, followed by a list of parameter settings, and ends with the keywords **end userparams**.
Each parameter setting has the following syntax:

<param-name> : "<description>" : <value> ;

For example:

```
userparams
    stperiod : "Short term period" : 0:05:00;
    mtperiod : "Medium term period" : 1:00;
    ltperiod : "Long term period" : 7 days;
end userparams
```

The three components in a parameter setting are separated by colons '**:**' and terminated with a semi-colon '**;**'. The first component is the parameter name. The second component is a string which describes what the parameter does. The third component is the value of the parameter.

NOTE:

1. The parameter value must have the proper *Alice* syntax for whatever quantity it represents. For example, times must use 24 hours and be in the HH:MM:SS format , share volumes must be in the x???? format, and prices must be in the $??.?? format. All the different formats are described in section 6.
2. The <param-name> can contain any character but must not begin with a number.
3. The 'userparams' section must go at the very top of the file (apart from comments).

3.1.1 Disable command

The disable command is used to ignore and not issue any alerts by the alert rule.
A disable command is specified in the *user parameter* section. It begins with the keyword **disable**, followed by a comma-separated list of alert codes. Only alert rules which do not relate any of the alert codes in this list, will be checked in the alert run.
For example:

```
userparams
    stperiod : "Short term period" : 0:05:00;
    mtperiod : "Medium term period" : 1:00;
    disable 101, 103, 105;
end userparams
```

3.2 Declared Variables

Following the *user parameter* section, an *Alice* program generally contains a *declared variable* section. This section introduces all new variables which will be used later in the *Alice* program. It is necessary to declare a variable before it is used so that *Alice* knows what type of variable it is.
In the past, the *declared variable* section began with the keyword **at start**, followed by a list of variables to declare, and ends with the keyword **end at**. Now, with the new version of Alice, both **at start** and **end at** are no longer necessary. You may now declare variables anywhere you wish.
Each variable to declare is specified using the following syntax:

declare <variable-name> : <type>

For example:

declare tradedays : number

If you wish to 'index' the variable, in other words use it as an array by 'indexing' it with various numbers or values, you use the following syntax:

declare turnover[security] : volume

declare profit[house] : percent

declare turnover[security, house] : percent

Note that this is different from previous versions of Alice. You now have to specify the exact types you will use to index variables with, in order to catch mistakes where the wrong types were used inside the '[]' brackets. Note also that you can re-use the same array name with different types, provided you separately declare each form of usage, for example as you see above with the 'turnover' arrays.
Refer to section 6 for a complete list of different 'types' available.
The declared variables shown above, are initialised to contain undefined values, by default. In other words if you tried to access these variables without assigning anything else to them first, you will get an **(undefd)** value. In some cases this may not be desirable.

The new Alice provides you with a choice of leaving the initialisation to undefined values, or change them to be initialised to zero values. To declare variables with zero initialisation values, you use the **zeroed** feature, shown below:

declare turnover[security] : zeroed volume

declare turnover[security, house] : zeroed percent

Note! The **declare …: zeroed** syntax does not clear out values, it merely changes how the program treats new elements of this array which it hasn't seen before. You cannot use a 'declare' statement to reset values. If you want to reset values, e.g. clear them out at the start of each day, you will need to use a loop and "let" statements.

3.3 Benchmarks

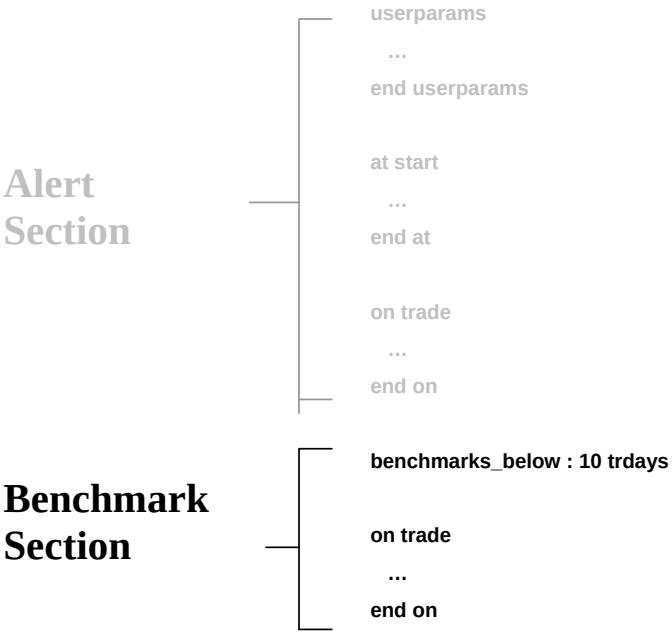
Alice has a special language feature to allow the user to compute benchmarks. This feature is called the "benchmarks_below" feature.

3.3.1 The "benchmarks_below" feature

An *Alice* program which uses benchmarks must be split into two halves: the code which performs the alerting, and the code which performs the benchmarking. Between these two halves is the following syntax:

```
benchmarks_below : <X>
```

For example:



When the alerting system runs, it will first run the benchmarks section (the lower section). Based on the example above, if the alert run is done on the 15/6/1997, then benchmarks will be run over the 10 trading days prior to this date ie: 1/6/1997-14/6/1997.

The number of days to go backward is defined after the semi-colon '**:**' of the **benchmarks_below** statement. This number can be made into a user parameter, or it can be specified exactly as in this example. If you use the word 'days' after the number, then it will refer to calendar days (e.g. 14 days = 2 weeks). If you use the word 'trdays', then it will refer to trading days (e.g. 10 days = 2 weeks, provided there are no public holidays in-between).

All clauses in the benchmarking section will be executed during the benchmark run, e.g. **at start**, **at daystart**, **on trade**, **every 5 mins**, etc. During this phase, the program collects benchmark statistics which will later be used as the thresholds in each alert rule. To communicate these thresholds to an alert rule, these thresholds must be assigned to arrays/variables which can either be declared at the beginning of the alert program or in the benchmark section.

3.3.2 When are benchmarks computed?

Because a benchmarking run can often take a very long time, **SMARTS** is capable of separating the two phases so that the benchmarking phase is run at the end of the previous day, and the alerting phase

begins immediately the following day. All the values of the arrays are stored in a 'benchmark file' for that date.
If the user requests an alerting run, and the benchmark file is up-to-date, then the alerting phase can begin immediately. However, if (a) the benchmarks did not run, or (b) the benchmark part of the program has been modified, or (c) the *userparams* which affect the benchmark phase have been modified, then the benchmark file will be considered 'out-of-date'. In this case, benchmarks must be computed before alerting begins. This computation can take a while.
For this reason, *Alice* programmers are discouraged from putting 'userparams' inside the benchmark code, in order to discourage users from modifying benchmark parameters. If a user modifies a benchmark parameter, this can invalidate a benchmark file and result in a significant delay before alerting can begin the following day. To avoid this delay, the process of modifying benchmark parameters is made slightly more difficult by requiring the *Alice* program to be directly modified.

3.3.3 Benchmarking Considerations

There are a lot of issues to consider when writing benchmarks.
The purpose of a benchmark is to separate normal trading behaviour, (usually everything below the benchmark) from abnormal trading behaviour (usually everything above the benchmark).
The statistical technique of 'standard deviations' can be used in some cases. For example, anything above 5 standard deviations can be considered abnormal. However, this measure can only be used with metrics which form a so-called 'normal distribution', or 'bell-curve'. Frequently, trading metrics such as trade-size do not form a normal distribution. If standard deviations were used in these cases to form benchmarks, users would find themselves raising the threshold for alerts from 5 to 50 to 100 standard deviations, in order to reduce the number of alerts to an appropriate level.
Instead, **SMARTS** recommends the use of 'empirical benchmarks', where benchmarks are taken by using histogram techniques on actual observed sets of values for metrics. For example, you can form the collection of all trade-sizes, and then take the top 0.5% of these values and call them abnormal. Or, you can take the top 1% of values and multiply by 10 and regard that as the cutoff point.
However, the disadvantage of this technique is that if you do not have enough numbers in your distribution, then the 1% cutoff values can be meaningless. The solution in this case could be to use 5% instead, and then multiply by an arbitrary number such as 100.
Or, perhaps all securities should be combined into a single distribution to get more numbers. However, this has the disadvantage that the benchmarks might be less appropriate to each security, for example benchmarks for liquid securities are being used to judge illiquid securities.
A longer benchmark period might also be useful. However, this has the disadvantage that the benchmarks are slower to adjust to changing market conditions.
When writing a benchmark, one should also consider what data should be included in the benchmark calculations. For example, should one day of especially heavy trading be allowed to affect benchmarks for the next 14 days? Should off-market trades be combined with on-market trades when creating benchmarks? Should the benchmark consider all securities as a group, or should it be security-specific, or somewhere in-between the two? For example, maybe securities can be grouped according to liquid vs illiquid securities.

3.4 Syntax of a complete Alice program

```
#------------- USER PARAMETERS SECTION ----------
    userparams
        <param-name> : “<description>” : <value>;
        <param-name> : “<description>” : <value>;
        <param-name> : “<description>” : <value>;
    end userparams


#------------- GLOBAB VARIABLE DECLARATIONS ---------
    declare <variable-name> : <type>
    declare <variable-name> : <type>
    declare <variable-name> : <type>

#------------- ALERT RULE -------------------
    <when clause>
        <statements - alert conditions>
            < statements - alert statement>
        <end statements>
    <end when clause>


#------------- BENCHMARK SECTION --------------
    benchmarks_below : <benchmark period>

    <when clause>
        distribution creation
    <end when clause>

    at end
        <per statement>
            cutoff point creation
        <end per statement>
    end at
```

3.5 An example of a complete Alice program

This *Alice* program is defined to issue an alert whenever a trade has a value greater than the top 1% cutoff level over the benchmark period.

```
userparams
    benchmark_period : "Length of benchmark interval" : 30 days;
end userparams

at start
    declare largetradeval : value
end at

on trade
    if value > 10 * largetradeval[security] then
        alert 101, "BIG TRADE", "A big trade of [value] occurred;
          benchmark=[10*largetradeval[security]]"
    end if
end on

benchmarks_below : benchmark_period

at start
    declare TradeVal : distribution
end at

on trade
    let TradeVal[security] <- value
end on
# for each security, create a TradeVal distribution.  Add to this distribution the value of each
# trade executed in the that security.

at end
    per security
        let largetradeval[security] = distcutoff(TradeVal[security], 99%) * $1
    end per
end at
# for each security, determine the top 1% of the value distribution and assign it to the variable
# 'largetradeval'.  We multiply by "$1" to convert a number to a value.
```

4. WHEN CLAUSES

The *when clause* is the first part of an alert rule. It defines when the rule is to be executed. The alternatives are:

Function	Description
at <time>	Execute the rule at the specified time-of-day. If the alerting program is run over multiple days, the rule will be executed once each day. For example : To evaluate a rule at 4pm only, the when clause to use will be: **at 16:00** <alert rule> end at
at start	Execute the rule once at the start of the run. If the **at start** when clause is specified in the Alert Section, the rule following the **at start** when clause will be evaluated at the start of the alert run (ie at 0:00:00). If the **at start** when clause is specified in the Benchmark Section, the rule following the **at start** when clause will be evaluated at the start of the benchmark run. For example: To print a statement at the start of the alert run and at the start of the benchmark run, you would say: a) in the Alert Section **at start** print "STARTING ALERT RUN" end at b) in the Benchmark Section **at start** print "STARTING BENCHMARK RUN" end at Refer to section 5 for more explanations about the **print** *statement*. To assign newly declared variables (such as today_vol or today_val) an initial value at the start of the alert run, you would say: **at start** let today_vol = x0 let today_val = $0 end at NOTE: the **at start** when clause can be specified in both the Alert Section and the Benchmark Section or in just either sections.

Function	Description
at end	Execute the rule once at the end of the run (ie at 24:00:00). Similarly to **at start**, the rule following the **at end** when clause will be evaluated either at the end of the alert run or the benchmark run depending on which section it is specified in. For example: To print a statement at the end of the alert run and at the end of the benchmark run, you would say: a) in the Alert Section **at end** **print "COMPLETING ALERT RUN"** **end at** b) in the Benchmark Section **at end** **print "COMPLETING BENCHMARK RUN"** **end at** Note: some functions involving the date will not work in an "at end" statement – use "at dayend" instead.
at daystart	Execute the rule at the start of each (trading) day (ie at 0:00:01). The **at daystart** is similar to **at start** in that a) it evaluates the rule following the **at daystart** when clause at the start of every day; and b) depending on which section the **at daystart** when clause is specified in, the rule will be evaluated at the start of every day in either the alert run or the benchmark run. The difference between the **at daystart** and **at start** would only be significant when the alert run and/or the benchmark run spans across multiple days. For example: **at daystart** **print "Processing [date]"** **end at** If the above example is specified in the Alert Section and the alert program is ran over multiple trading days in ALMAS, the print statement will be printed once at the start of each trading day. If the above example is specified in the Benchmark Section and the benchmark period contains more than one trading day, the print statement will be printed once at the start of each trading day. NOTE: the print statement will not be printed for non trading days.
at dayend	Execute the rule at the end of each (trading) day (ie at 24:00:00). Similarly to **at daystart**, the rule following the **at dayend** when clause will be evaluated either at the end of every day in the alert run or the benchmark run depending on which section it is specified in.
every <x> **minutes** **every** <x> **seconds** **every** <x> **hours**	Execute the rule every <x> minutes (seconds or hours), starting when the market opens and finishing when the market closes. For example: To print the time of the current minute followed by the best bid and ask price of every security, say : **every 1 minutes** **print "at [time]"** **per security** **print "[security] best bid = [bid], best ask = [ask]"** **end per** **end every** If the market opens at 10:00:00 and closes at 16:00:00, the above rule will print the best bid and ask prices of every security every minute beginning with 10:00:00 and finishing with 16:00:00.

Function	Description
	Refer to section 5 for more explanations about the **per** *statement*. An extension of this when clause is to restrict the time across which the when clause is evaluated. This syntax of this when clause is **every** <x> **minutes from** <time1> **to** <time2> **every** <x> **seconds from** <time1> **to** <time2> **every** <x> **hours from** <time1> **to** <time2> For example: To evaluate a rule every 10 minutes in the last hour of trading, say: **every** 10 **minutes from** 15:00:00 **to** 16:00:00 .... **end every**
on entord	Execute the rule after each time an order is entered. For example: To fire an alert each time a new order is entered between 10:00:00 and 16:00:00 at a price 10% greater than the previous close price, say: **on entord** **if time >= 10:00:00 and time <= 16:00:00 then** **if change(price,closeprice(-1)) > 10% then** **alert …** **end if** **end if** **end on** Refer to section 7 for explanations about the **time**, **change**, **price** and **closeprice** functions and section 5 for explanations about the **alert** *statement*.
on amdord	Execute the rule after each time an order is amended.
on delord	Execute the rule after each time an order is deleted.
on trade	Execute the rule each time a trade occurs. The **on trade** when clause includes both on-market or off-market trades.
on mkttrade	Execute the rule each time an on-market trade occurs.

Function	Description
on offtrade	Execute the rule each time an off-market trade occurs.
on deltrade	Execute the rule each time a trade is deleted.
on amdtrade	Execute the rule each time a trade is amended. Trade amendments only involve the modification of minor attributes of the trade: amf, littlefields and tagfields.
before entord **before amdord** **before delord** **before trade** **before mkttrade** **before deltrade**	Execute the rule *before* the orderbook has been updated with the current transaction. For example: **before entord** **if is_bid and price > bid then** **print "New buy-price established at [time]."** **end if** **end before**
after entord **after amdord** **after delord** **after trade** **after mkttrade** **after deltrade**	'**after**' is simply an alias for '**on**'.
on index	Execute the rule each time an index update message arrives.
on control	Execute the rule each time a security/set of securities/trading engine changes status (e.g. suspended, open, etc.)
on info *(may not be available*)	Execute the rule on each news-wire or other text-information event.
on quote *(only applicable to quote driven markets*)	Execute the rule each time a bid/ask quote is input.
on pendingord *(Only applicable to markets with unconfirmed orders)*	Execute the rule each time an unconfirmed order is entered. An 'unconfirmed order' is simply an order from a junior trader which requires confirmation by a senior trader before being placed in the orderbook. Such orders will be visible in the orderbook but will not affect best bid/ask prices nor liquidity metrics and will not trade until they are confirmed. The entry confirmation of the order can be caught as an "on entord" transaction. Unconfirmed orders are treated by SMARTS as being 'non-real'. This is why the original "enter unconfirmed order" transactions don't come through in "on entord" clauses, but the "confirm order" transactions do.
on refupdate	An update to one of the reference databases (security.ref, broker.ref or trader.ref) has occurred. For example, a new instrument might have been listed. Exactly one of the following objects will be defined: security, house or trader, depending on which type of update it is. If it is not a new entity but rather just updates to fields, then there is no direct way to determine which fields have been changed. However, there is an indirect way, for example: **on refupdate** **if defined(security) and** **securityfield(security,"TY",date) != securityfield(security, "TY", date-1) then** **print "The TY field has changed"** **end if** **end on**

5. STATEMENTS

Statements do not need to be terminated by a semi-colon or other symbol, as in C or Pascal, because the keyword of the following statement is enough information to determine where a statement ends.

Function	Description
alert	Defines the expressions to attach to an alert when fired in **ALMAS**. Begins with the keyword **alert**, followed by a list of parameters. The parameters determine the fields of the alert. A basic **alert** *statement* contains the following components: **alert** <code>**, "**<shorttext>**", "**<full text>**"** where <code> — a numerical expression representing the alert code. <shorttext> — a brief description of the alert. <full text> — an extended description of the alert. This string may extend over multiple lines. For example: **alert 801, "ST VOL", "LARGE SHORT TERM VOLUME over the past hour"** **Embedded expressions in alert statements:** More often than not, instead of just printing an extended description of the alert, you might also want to print additional information derived from the trading activity such as the actual volume of turnover or the last trade price and so forth. To do this, you would simply include the desired expression between square brackets '**[ ]**' in the <full text> component of the **alert** *statement*. When the alert containing these embedded expressions is fired, the embedded expressions will be replaced with the underlying values. NOTE: An expression can be any *Alice* function or other variables defined in the *Alice* program. For example: **alert 801, "ST VOL", "LARGE SHORT TERM VOLUME over the past hour was [volume(1 hours)]"** When this alert is fired, the expression **[volume(1 hours)]** will be replaced with a number representing the total volume traded over the past hour. **alert 901, "LGE TRADE", "LARGE TRADE : [volume] at [price] ([value]) between [buyerh] and [sellerh]."** When this alert is fired, all the expressions between square brackets will be replaced with their respective values. **Optional fields in alert statements:** In addition to the above three components of an **alert** *statement*, there is also a fourth component known as the optional fields. These fields determine a) when an alert should be reissued; b) the intensity of the alert; c) what additional information to display in ALMAS; and d) what information to send to other SMARTS applications when launching from ALMAS. See section 0 of this manual for more information about these optional fields.

Function	Description
break **(from v6 only)**	Exit a 'for' or 'per' or 'for each' loop prematurely.
declare	Introduces a new variable for future use in the *Alice* program so that the system knows what **type** it is. The syntax used to declare a new variable is: **declare** <variablename> **:** <type> where <variablename> a name given to the new variable. This name must not contain any spaces but it can consist of multiple words joined by an underscore. <type> a variable can take the form of a security, date, time, price, value, volume, house, trader, client, number, boolean, string or percent. Refer to section 6 for a more detailed explanation of each of these **types**. For example : **declare last_trade_time : time** introduces the new variable **last_trade_time** which is going to take on a 'time' value. **declare tmp_val : value** introduces the new variable **tmp_val** which is going to take on a 'value' value. **NOTE**: A new variable must be declared before being used. A new variable cannot be assigned to more than one **type**. Generally all new variables are declared at the start of the *Alice* program but they can be declared anywhere in the *Alice* program such as the start of the alert rule in which the new variable is being used. The benefits of declaring all new variables in one common spot is so that you won't forget what new variables you have already created and thus should not reuse. Once a variable has been declared, it can then be assigned a value and used anywhere else in the *Alice* program. It is possible to assign a different value at different parts of the *Alice* program. If this is the case, the variable will always take the value it was last assigned to. For example: **at start** **declare tmp_val : value** **end at** **every 5 minutes** **let tmp_val = $200** **if …** **alert 101,"…** **end every** **every 30 minutes** **let tmp_val = $400** **if …** **alert 201,"…** **end every** In the above *Alice* program, the variable **tmp_val** will always take the value $200 when evaluating alert 101 and value $400 when evaluating alert 201. **Declaring and assigning values to variables in one statement:** A special form of **declare** statement allows you to both declare and assign a variable to a

Function	Description
	value in one statement. The syntax used to do this is : **declare let** <variablename> **=** <assigned value>
	For example: To declare and assign a value to the **tmp_val** variable in the previous example in one statement you would a) not have to specify the **at start … end at** statement; but instead b) would combine the **declare** statement with the first **let** statement used to assign the variable a value. Note, when combining the **declare** and **let** statements, it is not necessary to tell *Alice* what <type> the variable will represent. **every 5 minutes** **declare let tmp_val = $200** **if …** **alert 101,"…** **end every** **every 30 minutes** **let tmp_val = $400** **if …** **alert 201,"…** **end every** **NOTE**: Limitations with respect to declaring and assigning values to variables in one statement: 1. Can only be done with variables which are assigned a fixed number at any one time. ie it cannot be done when a variable is continuously added to, such as **on trade** **let cum_val += value** **end on** In the above example, the variable **cum_val** is continuously added to after every trade, thus it is not possible to declare and assign this variable a value in one statement. 2. If the variable is used in multiple rules which are evaluated at different times of the day, the variable must be declared the first time the variable is used in a rule. **Using declared variables as an array:** An array is a variable which is indexed by a particular characteristic such as security, house and so forth. The syntax used when assigning a variable values whilst the variable is indexed by an array is **let** <variablename>**[characteristic] =** <assigned value> Note, it is not necessary to specify the characteristics by which a variable will be indexed on when declaring the variable. It is only necessary when assigning the variable values. For more information about using variables as an array, see the description of the **let** *statement*.
end	The word end is used to terminate every *when clause* and every *statement* which encloses other *statements*. It must be followed by the keyword which it is terminating. It is not a statement in the usual sense, because it is merely part of other statements. Examples: **if** 3 < 4 then print "correct" else print "incorrect" **end if**

Function	Description
	per security ... **end per** **with** ^BHP ... **end with** **on** trade ... **end on**

Function	Description

for	A **for** statement is used to perform loops in the *Alice* program, that is, perform a set of statements repeatedly. The syntax is:
	for <initialise> **;** <condition> **;** <increment> **do** <body> **end for**
	where
	<initialise> refers to some *statement* executed once at the beginning of the loop;
	<condition> refers to some boolean expression : the loop will continue as long as this condition is true.
	<increment> refers to some other *statement* used to increment the loop counter in some way.
	<body> refers to the *statement* (or set of statements enclosed in { } braces) which the user wishes to have executed repeatedly.
	
	For example,
	To print the numbers 0 to 9, the *Alice* code would say:
	`declare x : number`
	
	`at start`
	`for x = 0 ; x < 10 ; x = x + 1 do`
	`print "x"`
	`end for`
	`end at`
	
	NOTE: Usually **for** *statements* are used mainly for looping over numbers and dates. To loop over securities, houses traders and orders, the **per** statement should be used instead.
	NOTE 2: To break out of a loop prematurely, use the **break** keyword (from v6 only).
	
	Using the for statement as a 'while loop':
	The **for** *statement* can also be used as a 'while loop', simply by leaving out the <initialise> and <increment> parts of the **for** *statement*, such that the syntax becomes:
	for ; <condition> **; do** <body> **end for**
	
	A 'while loop' would generally be used when more complex <increment> values are to be used in the rule. For example:
	To print every number between 0 to 9 followed by every 10th number between 10 to 100 you would say:
	`declare x : number`
	
	`at start`
	`let x = 0`
	
	`for ; x <= 100 ; do`
	`print "x"`
	
	`if x < 10 then`
	`let x = x+1`
	`elsif x <= 100 then`
	`let x = x+10`
	`end if`
	`end for`
	`end at`

for each
(from v6 only)
	"for each" lets you iterate over all members of an array, e.g.:
	`declare i : number`
	`declare j : string`

Function	Description
	... **for each** A[I,j] print "I=[I], j=[j], Aij=[A[I,j]]" **end for**
if	The **if** *statement* defines a test a condition followed by the appropriate actions to take should the conditions be met. The syntax of an **if** *statement* a) For a simple one condition rule **if** <condition> **then** <statement-list> **end if** b) For a two condition rule **if** <condition> **then** <statement-list> **else** <statement-list> **end if** If the condition is true, then it executes the statement after the keyword **then**. If the condition is false, and there is an **else** part, it executes the statement after the keyword **else**. For example: To issue a PRICE RISE alert when the change in price is positive and a PRICE FALL alert when the change in price is negative, you could say: on trade **if** change(trueprice,closeprice(-1)) > 20% **then** alert 101,"PRICE RISE","PRICE RISE : … **else** **if** change(trueprice,closeprice(-1)) < -20% **then** alert 111,"PRICE FALL","PRICE FALL : … **end if** **end if** end on Special care needs to be taken with the **end if** part when **if** *statements* are being included inside other **if** *statements*. For every **if** *statement* used, there must be a corresponding **end if** statement. **Simplifying multiple if statements:** Sometimes (especially when assigning values to a new variable), it may be clumsy to have to end every **if** *statement* used. In such situations, it is possible to combined all the **if** *statements* into one. The syntax for this would be: **if** <condition> **then** <statement-list> **elsif** <condition> **then** <statement-list> **elsif** <condition> **then** <statement-list> **else** <statement-list>

Function	Description

```
        end if
```

For example:

To assign a different value to a new variable depending on the rank of a security, you could say:

```
at start
    declare val_threshold : value
end at

at start
    per security
        if rank(date -7 days) <= 10 then
            let val_threshold[security] = $1000000
        elsif rank(date -7 days) > 10 and rank(date-7 days) <= 50 then
            let val_threshold[security] = $500000
        elsif rank(date-7 days) > 50 and rank(date-7 days) <= 1000 then
            let val_threshold[security] = $100000
        end if
    end per
end at
```

NOTE: Important to remember when assigning new variables a value is that if the condition is 'undefined', ie. it cannot be determined as either true or false, none of the **if** *statements* will be executed. To overcome this, you must therefore also take into account conditions which are undefined and assign a value to the variable.

For example:

To assign a new variable a value based on a security tag which takes the values "1", "2" or blank :

```
at start
    declare in_index : string
end at

at start
    per security
        if undefined(securityfield("IN")) then
          let in_index[security] = "N"
        elsif securityfield("IN") = "1" or securityfield("IN") = "2" then
          let in_index[security] = "Y"
        end if
    end per
end at
```

For more explanation of the 'undefined' mechanism see section 8.3.

The **if** statements is also used when the user wants to identify a specific list of objects. For example to only look at trading in securities CBTP, SMWY and FIEC, type

```
<when clause>
    per security
        if security = ^CBTP or security = ^SMWY or security = ^FIEC then
            .....
        end if
    end per
end <when clause>
```

Function	Description
	If the user only wants to identify a single object, either the **if** *statement* or the **with** *statement* (described later) can be used.

Function	Description
let	The **let** statement is used when assigning values to variables. The syntax is: **let** <variable> = <value> where <variable> is either a simple name, or an array expression (explained below). <value> is an expression of any type. = operator which lets the <variable> equal to the <value> Some other common operators used to assign values to the variables are : += The '+=' operator is used to accumulate the sum of many values in a variable. A statement of the form: "let X += Y" is similar to: "let X = X + Y". The only difference is that the second form will not work if 'X' is initially undefined. This is the major advantage of the '+=' form: X will automatically be set to zero if it has not been defined. <- The '<-' operator is used to insert values into a 'distribution variable'. A distribution variable is any variable declared as: <var> : distribution. These variables are used to perform statistics on a set of numbers, for example to compute the median or the top 1% cutoff value. See the 'distcutoff()' function in section 7.2. For a list of other operators, refer to section 8.2. **NOTE**: There are two important conditions to remember when using the **let** statement: 1. a variable must always be declared before it is used. 2. a variable must always store a <value> of the same **type**. For example: To assign the variable a single value **at daystart** **let tempval = $500** **end at** To add a value to the variable each time a trade occurs **on trade** **let cum_val += value** **end at** **Arrays:** What can an array consist of ? An array can only store values of one type. This means that all **let** statements for an array with a given name must have a value of the same **type** on the left-hand-side. The reason for this was so that *Alice* could do static type-checking (ie. identify type errors at compile-time). An array however can be index based on any characteristics or combination of characteristics. For example numbers, times, houses, securities or combinations of these. You are not limited to indexing arrays by integers. (Such arrays are called 'associative arrays'). If you want to index an array with multiple values, the values must be separated by commas. The syntax of an array is **let** <variable>**[characterisic]** = <value> or **let** <variable>**[characterisic1, characteristic2, …]** = <value> For example: **let tempval[house] = $500** # This **let** statement assigns a value of $500 to the **tempval** variable of each house.

Function	Description
	Let tempval**[house, security]** = $500 # This **let** statement assigns a value of $500 to the tempval variable of each house in each security. **NOTE**: An array indexed by different characteristics can have the same name.
per	A **per** statement allows you to use one of the built-in looping constructs without having to deal with the **for** statement. For example: To loop over one construct: Looping over all securities: **per** security Looping over all houses: **per** house Looping over all traders: **per** trader Looping over all clients: **per** client Looping over all indices **per** index To loop over multiple constructs using two **per** statements: Looping over all security and house pairs: **per** security **per** house ……. To loop over all security including the delisted or matured securities, the following per statement would be used: **per** all security To only loop over active security and house pairs, where 'active' means that this house has traded/entered an order in this security already on the *current day*, the following per statement would be used: **per** active security, house **NOTE**: It is important to remember that this only applies to a single day, not the full alerting period. This means that it cannot be used inside a **at end** construct, only **at dayend**. Also, it cannot be used inside an **at start** or **at daystart** construct, because at this point (i.e. before trading), all houses are regarded as inactive. **NOTE2**: A security or (security,house) combination is regarded as *active* if they have at least one **trade or order** in this stock up to the present time. The **per** statement is always followed by a series of **statements** which correspond to the code you want executed for each security/house/etc. At the end of this series of **statements** you must type **end per**. That code is executed repeatedly, with the current security / current house variables modified accordingly. For example, To print all securities with a trueprice above $20.00, you simply type: **per** security if trueprice > $20.00 then print "[security]'s trueprice is above $20.00" end if **end per** You can also use the **per** statement to loop over orders in the current order-book, e.g.: **per** order if buy_or_sell = "B" then print "[price], [volume]" end if **end per**

Function	Description
	This loops over all the orders, both bid and ask orders, in the current order-book for the current security. You must have the current-security defined in order to use this feature.
	Sorting via the **per statement** using the **sortby statement** The new version of Alice now has a sorting feature that you may use in conjunction with the **per** loop. To use the sorting feature, you need to use the **sortby statement**. The sorting is of course in alphanumeric order. The **sortby statement** requires a parameter to sort with, i.e. a field or category to sort by. For example, Possibly the most popular use of the **sortby statement** will be to sort outputs by security field "LN", the long names of each security. So, to print all the securities with a trueprice above $20.00, in alphabetical order of the security long name fields, you simply type the following: **per** security **sortby securityfield("LN")** **if trueprice > $20.00 then** **print "[security] [securityfield("LN")]'s trueprice is above $20.00"** **end if** **end per** **T**his will show on screen the previous output in alphabetical order of the long name field of each security.
print	A **print** *statement* allows you to print text and values to the Messages Window in **ALMAS** during the Benchmark and/or Alerts run. The syntax is: **print** "<value>" where: <value> can be any text or numeric expression , including numeric formulae. The format of the output depends on the **type** of value, for example, Prices are printed with a '$' and two decimal places. Volume amounts are printed with a preceding 'x' symbol without commas. value amounts are printed with a '$' and zero decimal places. For example, To print the volume, value and number of trades executed in each security at the end of the day, type details of trades executed price of the current security at 12:30pm (note that a twenty four hour clock is use in *Alice*) type: **at dayend** **per security** **print "[security] turnover today was :** **[volumeall(0:00:00)], [valueall(0:00:00)], [tcountall(0:00:00)]"** **end per** **end at** There are several thing to take note of in the above example: a) **print** *statements* may extend over multiple lines but the output would be displayed on the same line so long as the Messages Window in ALMAS is wide enough to display the full print statement; b) ordinary text do not need to be enclosed between square brackets [ ] (ie : turnover today was :); c) expressions which you would like replaced with the actual values need to be enclosed between square brackets (ie : [security], [volumeall(0:00:00)], etc); d) each expression to be replaced with the actual values must be enclosed in their own square brackets. They cannot be combined in one square bracket separated by

Function	Description
	spaces or commas, unless you intend to perform an operation between the two expressions. For example: [**volumeall(alertdate) / volumeall(-20)**] If an *Alice* file contains multiple **print** *statements*, by default each **print** *statement* will be printed on a line of its own ie with the newline character at the end. To print multiple **print** *statements* continuously instead of a new line for each new statement, type three dots after the **print** *statement* followed by another **print** *statement* with a newline character. For example: To print a list of active securities during the trading day, the following **print** *statements* will produce the following results: a) **at dayend** **per active security** **print "[security]"** **end per** **end at** This **print** *statement* will produce a list of active securities with each security displayed on a new line like so: IIPC SMWY CBTP b) **at dayend** **per active security** **print "[security] " ...** **end per** **end at** This **print** *statement* will produce a list of active securities with each security displayed one after the other: IIPC SMWY CBTP **NOTE**: Since every print statement will output one line of text, to insert an empty line in between each line of output or to simply print text on the next line without using another print statement, you may use the '\n' special symbol. The '\n', when placed within the text to be printed, will print the text following it on the next line. For example : a) **print "[security]\n"** results: IIPC SMWY CBTP b) **print "[security]\nwith long name : [securityfield("LN")]\n"** results: IIPC with long name : Interphase Solutions SMWY with long name : Swift Corp Co Inc CBTP with long name : CoVest Corp Inc

Function	Description
	Please turn to page 37, on **string** for more about special symbols like '\n'.
printto	Prints text and values to a file which can be retrieved and viewed via a text editor such as **ALDIT**. This is useful for exporting data from ***SMARTS*** and also for debugging from ***ALMAS***. The syntax is: **printto** "<filename>", "<value>" where: <filename> — the name of the output file to which the printed text will be dumped. You can select an absolute UNIX path-name, or a simple or relative path-name. If you select a simple path-name, the file will be created in the user's smarts directory, which is typically $HOME/smarts. **NOTE**: If the filename is assigned to a string variable, it is not necessary to have the quotation marks **"** and **"** around the <filename>. A filename would normally be assigned to a string if there are many **printto** *statements* and you don't want to repeat the filename every time. <value> — can be any string or other value calculated from the trading data. Typically it will be a string with embedded expressions enclosed in **[** and **]** brackets. If data is being exported, this string will often contain '**|**' marks or some other field separator such as commas '**,**'. For example: a) **at dayend** **per security** **printto "**TEST.csv**","**security [security], volume = [volume]**"** **end per** **end at** The file **TEST.csv** will be created when the first **printto** *statement* is executed, and thereafter appended to until the end of the run. Like the **print** *statement*, **printto** *statements* are normally printed on a new line of its own, ie. with the newline character at the end. If you don't want the newline character, type three dots after the command. But don't forget that you will eventually need a newline character – typically you'll have another **printto** statement after the loop.

Function	Description
	b) at dayend per active security printto "TEST.csv", "[security] " ... end per printto "Done". end at The first **printto** *statement* will produce a list of active securities with each security into the newly created file **TEST.csv**: IIPC SMWY CBTP The second **printto** *statement* will print the word Done after CBTP: IIPC SMWY CBTP Done And as a result of the full-stop (the dot) at the end of the **printto** statement, it will then close the file such that no file output handler will be kept and avoids the waste of system resource if no more printing is done to **TEST.csv**.
printcsv	Similar to the **printto** statement, this statement prints text and values to a file, but unlike the **printto**, this **printcsv** statement is tailor made for printing in csv/sdv/txt format to a file (usually for upload into Excel). Using the **printto** statement, you would have to embed the values that you want to print inside a string. But using the **printcsv** you would only need to give it a comma separate list of values, without square brackets. For example: At dayend per security printcsv "TEST.csv", security, securityfiled("LN"), volume, trueprice end per end at The above code would print to the file TEST.csv the current security code, security long name and the security trueprice. Notice how **printcsv** automatically outputs the values to the file in a format that Excel will like, stripping out the 'x' in volume, but leaving the '$' sign for price, blank fields for undefined values. So, you no longer need to use **format** to get values output properly to a file. Note that there are actually 3 file formats supported, and the format used depends on the extension on the filename used: ***csv*** Comma-separated values (i.e. a comma between values) ***sdv*** Semi-colon delimited values (i.e. a semicolon between values) ***txt*** Tab-separated values (i.e. a tab between values) Note: Some fields such as house long names and security long names may contain commas e.g. "Were Stockbroking Limited, Trading as …" The comma will split the output over two cells unless the user saves as an *.sdv or *txt file. Note that the use of a single dot to close the written file can apply to **printcsv** statement as well.
printdist	Similar to printcsv, except more suited to printing distribution variables. It is designed to print to a file with a ".dist" extension, but should work with any file. The output file will be similar to a .csv file, except instead of comma delimited, values are delimited by '\1' character. This distinguishes the file content during read (with the read command – see below).

Function	Description

	For example, to print a distribution to a file:
	`declare valueDist: distribution`
	
	`on trade`
	`  valueDist <- value`
	`end on`
	
	`at dayend`
	`  print "[distcount(valueDist)] [distaverage(valueDist)] [valueDist]"`
	`  printdist "test.dist", "t1", "t1.1", valueDist`
	`  printdist "test.dist", "t2", "t2.1", valueDist`
	`end at`
	
	And then, to use what was printed above, we use the read command as per usual (see below for more information on the read command):
	`declare valueArray[string, string]: distribution`
	
	`at start`
	`  read "test.dist", string, string, valueArray`
	`  print " [distaverage(valueArray["t1", "t1.1"])] [valueArray["t1", "t1.1"]]"`
	`  print "[distaverage(valueArray["t2", "t2.1"])] [valueArray["t2", "t2.1"]]"`
	`end at`
	
	The printdist command is most useful to complement tasks that requires the storage of distributions for later use. For example: rolling benchmarks.

read	This statement allows you to read data from a csv/sdv/txt file and store the values in Alice arrays.
	The syntax is:
	read <filename>, <column1>, ...
	where:
	<filename> is a string expression that specifies the file. The file must be a text file with fields separated by either a comma (csv files), semicolon (sdv files) or a tab (txt files). You should either specify the full path name or the path relative from your 'smarts' directory (or the current directory in the case of the command-line usage of alertserver).
	<column1>, ... is a list of columns. Each 'column' is represented by an identifier. Each column is regarded as either a 'key' column or a 'value' column (or blank meaning ignore this column). The 'key' fields become the indexes into the arrays, and the 'value' columns name the variables or arrays which the data is stored into.
	
	For example, if foo.csv contains:
	`CBTP,25/12/2000,1000,44%,hellomate`
	
	then this program:
	`at start`
	`  declare my_volume[security,date] : volume`

Function	Description
	declare my_value[security,date] : value declare my_percent[security,date] : percent declare my_string[security,date] : string read "foo.csv", security, date, my_volume, my_percent, my_string print "[my_volume[^CBTP,25/12/2000]], [my_percent[^CBTP,25/12/2000]], [my_string[^CBTP,25/12/2000]]" end at will output this: `x1000,44%,hellomate` *Tip:* To ignore a column, simply leave it blank in the 'read' statement, i.e. have the next comma follow straight on from the previous comma. You can also use the word 'dummy' to the same effect, e.g.: read "foo.csv", security,date,dummy,my_percent,my_string *Tip:* At some sites, a security-code can mean different securities on different dates. This can happen for example if the code ABC is used by one company which goes bust and is later re-used by another completely different company. In this case, the string 'ABC' in your file will be ambiguous – it might not be clear which ABC you are referring to. Internally, SMARTS keeps the companies separate by assigning a unique 'mpl-id' to each. If you want to output a file of intermediate results to be read in by another Alice program, then perhaps you should output the mpl-id instead of the security-code e.g.: printcsv "intermediate.csv", "#[mplid(security)]", turnover[security] instead of: printcsv "intermediate.csv", security, turnover[security] This will cause your file to contain eg.: #131,$10542170 instead of: ABC, $10542170 When the next Alice program tries to interpret the "#131" as a security-code, it recognises the hash symbol as meaning that the mpl-id follows directly and therefore gets the intended company even if there is confusion over which company has the code 'ABC' on that particular date.
with	This statement allows you to artificially set the current security/house/trader etc. You might want to set the current security/house etc in order to get certain functions to work on a specified object. The syntax is: **with** <object> <statement-list> **end with** where: <object> can be a specific object such as a particular security or it can be an expression. Only one object may be specified at any one time. If multiple objects are to be specified, the user would use the **if** statement instead. <statement-list> the commands to execute if the object or expression matches the <object> in the **with** *statement*. For example: To evaluate the amount of trading done just in security CBTP type: **with** ^CBTP if avdailyval > $1000 then print "CBTP has big trades" end if

Function	Description
	end with To determine whether the house who traded the most is a primary house type: **with** major_buyerh if primaryhouse then print "The main buyer is a primary house." end if **end with**
event	Outputs an event for analysis in the ***SMARTS*** research application called ***EVENTS***. This command appears on its own, ie. with no parameters. It should only appear in files run from ***EVENTS*** and will be ignored by ***ALMAS*** when generating alerts. For example: on trade if volume > x1000000 then **event** end if end on This command will generate a list of events which consists of the security code, date, start time and end time of every trade with a volume greater 1,000,000 units.
debug	Enters an interactive debug mode. In this mode, you can type Alice statements. You can enter any syntactically valid Alice statement and it will immediately be executed. Typically, you use it to examine the contents of variables using the 'print' statement, e.g.: **print x[security]** In fact, since 95% of statements entered in debug mode are 'print' statement, we have provided the abbreviation 'p' as an alias for 'print': **p x[security]** Note: If you want to print a single value, you don't need to use the double-quotes and square-brackets notation. For example, the following is a waste of typing: **p "[x[security]]" // ← don't type this.** Note 2: you can't enter 'when' clauses, since you are assumed to already be within a 'when' clause. The following will give an error if typed in debug mode: on trade p volume end on To continue from debugging, simply hit ENTER, i.e. enter a blank line. The program will then continue until the next 'debug' command.
exit	Immediately terminates the alerting engine. **Warning:** this command should *not* be used in the official Alice file, for obvious reasons.

free	Frees a whole array at once. For example:
	`declare A[security] : number`
	`declare A[security,number] : number`
	…
	`free A[]`
	The above statement will clear out all elements of all forms of the array 'A[]'. Immediately following this command, the value of A[<anything>] will be undefined. The memory used to store this array will be freed.
	The purpose of this command is primarily to help you avoid running out of memory, and secondarily to help you clear your array ready for the next day's processing or the next phase of processing. It is very common to free arrays at the end of a day's processing.
	Note: You must put empty square brackets: [ ] after the array name.
	Note: It is *not* possible to selectively free only certain forms or certain rows/columns of the array.

Function	Description
	Note: This method of freeing arrays can be used for single variables (i.e. variables without any square brackets or indexes) however this is very inefficient use of the statement. Also, small arrays without many elements are not efficiently freed using this method. To clear/free individual variables or small arrays, you should assign undefined values to the elements individually, e.g. using the '`dummyvolume/dummysecurity/dummydate`/etc' functions.

5.1 Alert Statement Fields

Below is the full list of the compulsory and optional fields which may be attached to an **alert** *statement*.

Field	Description	Type	Comments
Automatically generated fields			
date1	The date of the alert	date	Generated by system
time1	The time of the alert	time	Generated by system
security	Which security is it for?	security	Generated by system
Compulsory fields which must be specified in the alert statement			
code	What type of alert it is (which rule generated it)	number	Compulsory It is almost always a numerical constant.
shorttext	What is the name of the alert?	string	Compulsory
full text	What is the text of the alert?	string	Compulsory
Optional fields which can be added to the alert statement			
date0	The date the alert begins.	date	Optional By default it will be the same date as the **date1**.
time0	The time the alert begins.	time	Optional By default it will be the same time as the **time1**.
house	The house suspected of being involved.	house	Optional By default it will be **+all**.
trader *(may not be available)*	The trader suspected of being involved	trader	Optional By default it will be **+all**.
client *(may not be available)*	The client suspected of being involved	client	Optional By default it will be **+all**.
intensity	How significant/abnormal is the alert? How much did the current trading behaviour exceed the thresholds ?	number	Optional By default it is determined by the last if condition prior to the **alert** *statement*. See section 5.1.1 for other exceptions and more information.
viewer	Which **SMARTS** application to use when launching from ALMAS to visualise the alert.	string	Optional By default the visualisation application used is SPREAD.
commandline	Parameters to be given to the visualisation program.	string	Optional By default, uses whatever parameters that are set in relation to the alert. For instance date1, time1, security and any others. Other specific parameters not displayed in ALMAS may also be set in the Alice program using this commandline option. A complete list of parameters which may be defined in the commandline is provided in the Appendix A of this manual.
reissue	In what cases do we reissue an alert which is similar to a previous one?	string	Optional By default it is set to reissue the alert in the same security on the same date if the intensity increases by 10. See section 5.1.2 for more information
file	Attach a file to this Alert for analysis	string	Optional

	purposes.		By default no file is attached to an Alert.

The syntax of an **alert** *statement* containing optional fields is:

```
alert <code>, "<shorttext>", "<full text>",
        <optionalfield> = <values>,
        <optionalfield> = <values>,
        <optionalfield> = <values>
```

Recall that <optionalfield> not only determines what additional information to extract from the trading data but it also determines what parameters **ALMAS** should send to other applications. Five things about <optionalfield> to take note of:

1. The <optionalfields> = <values> string may extend over multiple lines.
2. If optional fields are attached to the end of an **alert** *statement*, remember to insert a comma "**,**" after the <full text> string.
3. If multiple optional fields are specified, separate each optional field by a comma "**,**". After the last optional field, it is not necessary to insert a comma.
4. <values> which are strings must be enclosed in quotation marks **"** and **"** .

For example:

To set the start date and start time of a long term volume alert to the beginning of the term,

```
alert 101, "LT VOL", "HIGH LONG TERM VOLUME: The volume over
  [lt_period] was [volume(date-lt_period)].",
        date0 = date - lt_period,
        time0 = 10:00:00,
```

To identify the house who did most of the trading during over the day,

```
alert 102, "LT VOL", "HIGH LONG TERM VOLUME: The volume over
  [lt_period] was [volume(date-lt_period)].",
        house = major_turnerh
```

To use REPORT as the viewer and to specify other specific parameters to set in REPORT,

```
alert 103, "PLENTY CROSSINGS", "PLENTY OF CROSSING TRADES:
   [num_cross[security]] by [num_houses]",
        viewer = "report",
        commandline = "-flags s+CR
                       -transactions TRADE
                       -showtrail no
                       -splittrades yes"
```

NOTE: The commandline options which can be used depends on which SMARTS application is set as the viewer. Appendix A of this Manual contains a complete list of all commandline options available for each SMARTS visualisation application.

5.1.1 The 'intensity' parameter

The intensity of an alert is a number, ranging from 0 to 100, which estimates the importance of an alert, or the likelihood that it does represent aberrant behaviour. Intensities are automatically generated by Alice but can be manually defined by the user.

5.1.1.1 Automatically Alice generated intensities

Most **alert** *statements* immediately follow an **if** statement. By default, intensities are thus generated based on the most recent **if** statement. This value is determined by 'fuzzy logic'. This fuzzy logic works as follows:
Each comparison operation yields a 'fuzzy logic' value. Suppose we have the expression X > Y. If X is not greater than Y, then the value will be 0, ie. false. If X is greater than Y, then the amount by which it is greater than Y will yield a value from 1-100. The formula is: $100 * \frac{X - Y}{X}$.

For example:

400 > 300 intensity = $100 * \frac{400 - 300}{400} = 25$

$500000 > $1000 intensity = $100 * \frac{500000 - 1000}{500000} = 99$

If the total trading since 10:00:00 is x1,000,000, the intensity of the following alert rule

```
if volumeall(10:00:00) > x500000 then
    alert 101,"HIGH TOVER","HIGH TURNOVER : [volumeall(10:00:00)] since 10:00:00"
```

would be $100 * \frac{1000000 - 500000}{1000000} = 50$

Occasionally, some alert rules may contain **if** statements which have more than one condition. If these multiple conditions are joined by an '**and**' operator, the intensity, which is determined by the fuzzy logic, will take the minimum value of these multiple conditions.
If these multiple conditions are joined by an '**or**' operator, the intensity, which is determined by the fuzzy logic, will take the maximum value of these multiple conditions.
For example

Say the total turnover since 10:00:00 was x1,000,000 and the change in trueprice to the previous day's closeprice was 25%, the following alert rule with two conditions joined by an '**and**' operator will have an intensity of :

```
if volumeall(10:00:00) > x500000 and
  change(trueprice,closeprice(-1)) > 20% then
    alert 102, …
```

intensity = $\min\left(100 * \frac{1000000 - 500000}{1000000}\ ,\ 100 * \frac{.25 - 0.20}{0.25}\right) = \min(50, 20) = 20$

With the same trading behaviour as above except that the conditions are joined by an '**or**' operator the alert will then have an intensity of :

```
if volumeall(10:00:00) > x500000 or
  change(trueprice,closeprice(-1)) > 20% then
    alert 102, …
```

intensity = $\max\left(100 * \frac{1000000 - 500000}{1000000}\ ,\ 100 * \frac{.25 - 0.20}{0.25}\right) = \max(50, 20) = 50$

5.1.1.2 Defining own calculation of intensities

Instead of letting Alice automatically generate an intensity value for an alert, the user can either provide their own intensity number or formula to be used by Alice to determine the intensity of the alert. The number or formula provided must return either a percent, a number or a boolean (ie true or false).

The number or formula to be used in determining the intensity of the alert will be specified following the **alert** *statement* optional field 'intensity'.
For example:

Say you have an alert rule with multiple conditions but you want the intensity to be always based on one of the conditions. To do this, you would specify in the optional intensity field which condition you would like Alice to always use when evaluating the intensity of the alert.

```
if time >= 15:30:00 and time <= 16:00:00 and
  change(trueprice,trueprice(-30minutes)) > 20% then
    alert 103, "DAY END PRICE RISE","DAY END PRICE RISE : …",
        intensity = change(trueprice,trueprice(-30minutes)) > 20%
```

To print the intensity of an alert using the **print** or **printto** statements, we first have to

1. declare a variable as a boolean; then
2. assign the **statement** used to determine the intensity to this variable.

For example:

```
at start
    declare printintensity: boolean
end at

on trade
    if volume(0:00) > x100000 then
        if volume > (volume(0:00) / tcount(0:00)) then
            let printintensity = (volume > (volume(0:00)/tcount(0:00)))
            print "There was an alert in [security] with an intensity of [printintensity]"
            alert .....
```

If all the alert conditions are met the following statement will be printed :
"There was an alert in … with an intensity of **truen%**"
where **true** = means that **volume** is greater than **volume(0:00)/tcount(0:00)**;
n% = represents the intensity calculated from the condition
(volume > (volume(0:00)/tcount(0:00)))

5.1.2 The 'reissue' parameter

When a given behaviour in the market causes an alert, it would normally cause many similar alerts on the same behaviour.
For example, a "short-term volume alert" may look at trading volume over the last hour, and it might be checked every 5 minutes. If an alert was issued each time this condition was checked and found to exceed the threshold, then there could easily be 10 or 20 alerts all alerting the analyst to the same security and time and the same type of behaviour. Alternatively, a "trade-to-trade price" alert which looks for large price jumps from one trade to the next might very often occur at the same time as a "short-term price alert" which looks at the price change over the last 5 minutes.
To prevent the analyst from getting too many redundant alerts, *Alice* has the 'reissue mechanism'. This mechanism attempts to suppress (as much as possible) redundant alerts, meaning alerts which are equivalent or very similar to previous alerts.
This mechanism works as follows:

- If two alerts have the same alert-code and the same security, then they are regarded as equivalent.
- If an alert arrives which is equivalent to a previous alert but with a much higher intensity (generally the intensity must be at least 10 points above the previous alert), then it 'supersedes' the previous alert. This means it overrides the previous alert so that the analyst will see only the second, more intense alert.
- If two alerts are considered equivalent and the second alert's intensity is not significantly higher, then the second alert is simply ignored and does not even enter the system.

The rules for when to consider two alerts 'equivalent' can be modified. For example, the analyst may want an alert issued for every (security, house) combination. The 'reissue' parameter which appears as an optional field in an **alert** *statement* is used for this. This parameter can also define what increase in intensity an alert needs before it supersedes a previous alert.
The value of this parameter is a string. The format of the string is as follows:

reissue = "<alertcode> <matching characters> + <increase in intensity>"

- <alertcode> - The first 3 characters are normally the alert-code of the first alert which identifies this particular behaviour. For example, if there are two alert-codes: 401 and 402, and they are very similar, then they will both have the number '401' at the beginning of the reissue string. This number does not have to be a valid alert-code, for example the number '40' could be used to denote all alerts in the 400-409 series. If there is no number here, the default is the current alert's alert-code.
- <matching characters> - Following the alert-code is a few characters which define whether two alerts need to match on security, house or whatever to be considered equivalent. For example

  'S' means they must match on security (+ alertcode) only;

  'SH' means they must match on security and house;

  'H' means they must match on house only;

  'ST' means they must match on security and trader;

  'SC' means they must match on security and client.

  Other combinations of these letters are also possible. If none of these letters are given, then all alerts that match on alertcode will be considered equivalent.
- + <increase in intensity> - Optionally, one can also add to the string the '+' character followed by a number. This number defines the minimum increase in intensity an alert needs to supersede another alert. This value is +10 by default, but can be any number from 0 to 100. Note that if the **alert** command is called repeatedly, with gradually increasing intensities, an alert will be created each time the intensity exceeds the intensity of the last alert by the required amount.

A string consisting only of the character '*' means issue the alert every time the program calls the alert command.
Examples of reissue strings are:

reissue = "*"	always issue the alert.
reissue = "305SH"	once per security, house pair unless intensity grows by 10 or more (ie the current intensity has to be equal to or greater than the previous intensity by 10). Remember by default the + <increase in intensity> is +10.
reissue = "305S+12"	once per security unless intensity grows by 12 or more.
reissue = "H+8"	once per house unless intensity grows by 8 or more.
reissue = "304"	once per alert-run for all alerts with alertcode=304.
reissue = "304D+-10"	once per day, unless the intensity is equal to or greater than the previous intensity minus 10. Say the previous intensity of an alert is 60. This alert will be reissued if the next alert has an intensity equal or greater than 50 (ie 60 - 10).

If no reissue parameter is specified, the default reissue parameter will be used. By default, an alert will be reissued once per security each day if the intensity increases by 10 or more, ie reissue = "SD+10"

6. TYPES

One feature of *Alice* is a type-system tailored to the needs of financial software.

Function	Description
security	This refers to a security. The security function returns the current security, and has this type. A security can be explicitly denoted with the '**^**' symbol followed by their security-code. Examples: **^CBTP**, **^SU002300456**, **^BHP**, **^0005**, **security**
house	This refers to a house. Most commonly, a house value comes from the house function, which returns the current house. A house can be explicitly denoted with the '@' symbol followed by their identifying code. Examples: **@B297**, **@C099090**, **@12465**, **major_turnerh**, **house**
trader *(may not be available)*	This refers to a trader. Most commonly, a trader value comes from the trader function, which returns the current trader. A trader can be explicitly denoted with the '**&**' symbol followed by their identifying code. Examples: **&MC0000201902**, **buyert**, **sellert**, **trader**
client *(may not be available)*	This refers to a client. A client can be explicitly denoted with the '`' (back-quote) symbol followed by their identifying code. Examples: **`SC987686330**, **buyerc**, **sellerc**, **client**
date	Date values. Example: **30/6/1995**. Arithmetic operations can also be performed on dates. For example: To determine the number of days between two dates, type: **30/6/1995 - 13/6/1995** To add/subtract a number of days to/from a date, type: **30/6/1995 + 14** **30/6/1995 + 14 days** The keyword '**days'** can be used to emphasise that a value is a date-difference value. **NOTE**: The number of days added (such as the "+ 7" in the examples above) includes weekends and any holidays, ie they refer to calendar days as oppose to trading days.
time	Time-of-day values. Times are specified in 24 hour format ie **HH:MM:SS** or **HH:MM**. Arithmetic operations can be performed on times. The keywords **minutes**, **seconds** and **hours** can be used to construct time values. - 1 hours means 1 hour beforehand. Examples: **15:45**, **10:00 - 0.5 hours**, **5 minutes**
price	Unit prices. Prices are specified with **$** signs and **two decimal points**. Prices are different from values, which store multiples of price times quantity. Examples: **price $4.50**, **5 * $1.25**
value	Quantities of money. Values are specified with **$** signs but have **no decimal point**. Commas or other separators should not be used. Examples: **$45000**, **0.5 * $45000**
volume	Number of units traded. Volumes are specified with the **x** symbol and must not contain any decimal points, commas or separators. Examples: **x10000**, **$45000 / $4.50**

Function	Description
number	Ordinary, dimensionless floating-point numbers. Useful for ratios and scaling factors. Examples: **3.33**, **$4.00/$.10**, **3**
percent	Percent, ie. the expression of a fraction like .34, as 34%. A percent constant is a number followed by a '%' sign. When it is multiplied by other quantities, it acts as a fraction (eg. 5% is equivalent to 0.05). Examples: **5%, percent(value, value(date))**
index	An index is usually used to identify market-indicator indexes, but it can also be used to identify general time-series such as interest rates. Examples: **X:DOWJONES**, **X:ALLORDS**
boolean	True and false values. The result of comparison operators and logical operators (and, or, not) is a boolean. They are used as the condition inside **if** statements. These values are actually fuzzy logic values, which is of relevance to the 'intensity' parameter of alerts. Examples: **true**, **3 < 4**, **false**, **price < $10.00** and **3 <= 4**
distribution	A distribution stores a collection of numeric values for doing histogram-type statistics on. It is used for computing medians, 1% cutoffs and so on. Refer to section 7.2 for a list of distribution functions.
string	A piece of text. A string is any text in-between double-quotes (**"**). As in C, any embedded \n, \t and \b symbols will be translated to newline, tab and backspace characters respectively. In addition, any string occurring anywhere in the program can contain **[expression]** symbols embedded in it. Wherever these occur, the expression is evaluated and the result placed in the string at the position of the **[ ]**. Examples: **"hello blind freddy"**, **" "**, **"there was an alert at [time]"**

6.1 Rules for mixing Types

1. Ordinary numbers such as 3.5 can be used in multiplication with any numerical quantity, ie. **numbers**, **percents**, **prices**, **values**, and **volumes**. Also, they can be used as the denominator in divisions with numerical quantities.
2. Any of the above numerical types can also be multiplied (on the right or left) by a **percent**.
3. Ordinary numbers with no decimal places can be added to and subtracted from **dates**: they imply a given number of days.
4. **Times** can be added to or subtracted from **times**. The result is a **time**.
5. **Prices**, **values** and **volumes** can be added or subtracted from things of the same type (but not things of different types).
6. A **price** multiplied by a **volume** gives a **value**. A **value** divided by a **volume** gives a **price**, and a **value** divided by a **price** gives a **volume**.
7. The **boolean** type stores fuzzy logic values. The programmer will only notice this feature in relation to the 'intensity' parameter of alerts. The details of this are explained in section 5.1.1 on the intensity parameter in the **alert** *statement*.

7. FUNCTIONS

A function is any word from the following list, which may or may not be followed by a list of parameters in brackets. For example:
date : Returns the date which is being examined for alerts. It does not need any parameters.
change(f, g) : Returns a ***percentage*** corresponding to how much '**f**' is larger than '**g**'.

The functions provided are those found to be the most useful microstructure measures for alert generation. There is no facility provided for the user to add more functions, but they can be added as necessary during the customisation period of an installation. New functions can also be provided as part of the maintenance of the software, in the form of *Alice* upgrades.
The tables that follow provide a full list of the available *Alice* functions.

7.1 Maths functions

X and Y below can be a **price**, **value**, **volume**, **percent, number** or **index** but number can only be a **number**.

Function	Description
abs(X)	Takes the absolute value of X. For example: If the current trueprice is $25.00 and the previous day's closeprice was $30.00 the following function will return : **abs(**change(trueprice,closeprice(-1))**)** = abs($\frac{25.00 - 30.00}{30.00}$ x 100%) = abs(-16.67%) = 16.67%
ceiling(X)	Rounds X up to the nearest integer. For example: **ceiling(**$141.500**)** = $141 **ceiling(**-575.50%**)** = -575% **ceiling(**-0.5**)** = 0 **ceiling(**7360.71**)** = 7361
change(X, Y)	Determines how much X increased or decreased from Y. ie $\frac{X - Y}{Y}$ x100% . Returns a **percent** type. For example: If the volume traded so far today was x14,622 and the average volume traded over the previous 5 trading days was x5,000 units, the following function will return : **change(**volumeall(0:00:00),(volumeall(trday(-6)) - volumeall(0:00:00))/5**)** = $\frac{14622 - 5000}{5000}$ x 100% = 192.44%
exp(X)	Returns e to the power of X where e is 2.71828. Returns a number. For example: **exp(**1**)** = 2.71828 **exp(**2**)** = 7.389 **exp(log(**2**))** = 2

Function	Description
float(X)	Converts X into an ordinary **number**. Also works with **dates**, **times** and **string**'s (provided the string is in decimal ASCII format). For example: **float(**$141.500**)** = 141.5 **float(**$917,589,389**)** = 9.17589e+08 **float(**x1,812,187,182**)** = 1.81219e+09 **float(**-50%**)** = -0.5 **float(**18/09/1998**)** = 11575 # When the float function is applied to a date, *Alice* will convert the date to a number representing the number of calendars days between the date and the 10/1/1967. **float(**0:00:55**)** = 55 # When the float function is applied to a time, *Alice* will convert the time to a number representing the number of seconds between the time and the start of the day ie 0:00:00.
floor(X)	Rounds X down to the nearest integer. For example: **floor(**-$141.500**)** = -$142 **floor(**-575.50%**)** = -576 **floor(**-0.5**)** = -1 **floor(**7360.71**)** = 7360
max(a,b)	Returns the maximum of any 2 numbers,percents,prices,values,volumes,dates OR times
min(a,b)	Returns the minimum of any 2 numbers,percents,prices,values,volumes,dates OR times
	WARNING! The above 2 functions could render some existing alice files incompatible with v5; if anyone has a variable called 'min' or 'max' then this will now clash with these new functions. The easiest solution is simply to rename the variables to something else e.g. "mymin" and "mymax"
percent(X, Y)	Determines what percentage is X of Y. ie. $\frac{X}{Y}$x100% . Returns a **percent** type. For example: If the volume traded so far today was x14,622 and the security has x2,000,000 units issued, the following function will return : **percent(**volumeall(0:00:00),issuedunits**)** $= \frac{14622}{2000000} \times 100\% = 0.73\%$
pow(x,y)	Returns x to the power of y
round(X)	Rounds X up or down to the nearest integer (up if fractional component greater than or equal to 1/2). For example: **round(**1234.75**)** = 1235 **round(**1234.50**)** = 1235 **round(**1234.25**)** = 1234 **round(**-2.5%**)** = -3% **round(**-575.50%**)** = -576% **round(**-150%**)** = -150%
sqrt(X)	The square root of X
log(X)	The natural log ie. $\log_e(X)$
percent(number)	Converts a number to a **percent**. ie. number x 100% (Note the scaling: for example, the number '1' becomes 100%.) For example: **percent(**0.5**)** = 50% **percent(**10**)** = 1000%

Function	Description
price(number)	Converts a number to a **price**. ie. number x $1.00 For example: **price(0.5)** = $0.50 **price(10)** = $10.00
sigmoid(number)	Puts X (a number) through the sigmoid function 100 * x/ (x+1).

NOTE: There is no equivalent to the **percent(number)** and **price(number)** for volume and value because the functions **volume(number)** and **value(number)** actually determines the total volume and value of trading over the number of days specified. See section 7.6.

7.2 Distribution functions

A distribution can store a collection of any numeric values, e.g. **price**, **value** and so on. Distributions are generally used in the benchmark section for computing benchmarks, but they can also be used to form a distribution on the current date.

Important: Distributions are explicitly forbidden from being carried forward from a benchmark phase to an alerting phase. This is because distributions use a lot of memory and this doesn't work well with benchmark cache files, and so instead we want to encourage people to store the relevant aspects of the distribution (max, average, cutoffs etc.) rather than the full data of the distribution.

When a numeric value is added into a distribution, it will be converted into a **number**. When using any of the Distribution Functions below (apart from **distcount(D)**), to identify a certain numeric value from the distribution it maybe necessary to convert this **number** back to the desired **type**. To do this, you would multiply the number by

- $1 to convert to a **value**;
- $1.00 to convert to a **price**;
- x1 to convert to a **volume**; or
- 1% to convert to a **percent**.

Function	Description

distcutoff(D, p)	Identifies the **p**% cutoff of the distribution **D**.
	The **p**% cutoff will be accurate to two significant digits.
	D must be a *distribution*, and **p** must be a *percentage*.
	For example:
	To create and identify the 99.95% cutoffs of the individual trade volume (over the past 30 days) distributions for the top 10 securities, you say:
	`at start`
	`  declare ind_trade_dist : distribution`
	`  declare ind_trade_cutoff : volume`
	`end at`
	
	`benchmarks_below : 30 days`
	`# To create a distribution`
	`  on trade`
	`  if rank(date-30 days) <= 10 then`
	`  let ind_trade_dist[security] <- volume`
	`  end if`
	`  end on`
	
	`# To identify the 99.95% cutoff`
	`  at end`
	`  per security`
	`  if rank(date-30 days) <= 10 then`
	`  let ind_trade_cutoff[security] =`
	`  distcutoff(ind_trade_dist[security], 99.95%) * x1`
	`  end if`
	`  end per`
	`  end at`
	
	This will return the top 0.05% volume from each of the top 10 securities' volume distribution.

Function	Description
	NOTE: **distcutoff(D, 50%)** will give the median of **D**

Function	Description

distaverage(D)	Identifies the average of the distribution.
	For example:
	To identify the average of the distribution above, say
	`at end`
	`per security`
	`if rank(date-30 days) <= 10 then`
	`let ind_trade_avg[security] =`
	`distaverage(ind_trade_dist[security]) * x1`
	`end if`
	`end per`
	`end at`

distcount(D)	Identifies the number of observations.
	For example:
	To identify the number of observations from the distribution above, say
	`at end`
	`per security`
	`if rank(date-30 days) <= 10 then`
	`let ind_trade_obv[security] =`
	`distcount(ind_trade_dist[security])`
	`end if`
	`end per`
	`end at`

diststdev(D)	Identifies the sample standard deviation (with the N-1 denominators).
distvariance(D)	Computes the variance of the distribution.
distmin(D)	Identifies the minimum observation.
distmax(D)	Identifies the maximum observation.

distmin(D,n)	Identifies the n'th smallest observation accurate to two significant digits.
	For example:
	To identify the 5'th smallest observation in the distribution above, say
	`at end`
	`per security`
	`if rank(date-30 days) <= 10 then`
	`let ind_trade_min_5[security] =`
	`distmin_5(ind_trade_dist[security],5) * x1`
	`end if`
	`end per`
	`end at`

distmax(D,n)	The n'th largest observation accurate to two significant digits.
distmedian(D)	The median value of this distribution. It is accurate only to 2 significant digits (this is for optimisation reasons). If there are two 'middle' scores, it takes the higher of the two.

distsum(D)	The sum of all values in the distribution.
	For example:
	To identify the sum and average of the distribution above, say
	`at end`
	`per security`
	`if rank(date-30 days) <= 10 then`
	`let sum[security] = distsum(ind_trade_dist[security])*x1`
	`let avg[security] = distaverage(ind_trade_dist[security])*x1`
	`end if`
	`end per`

Function	Description
	end at **NOTE**: This function should only be used if a distribution has already been created for some other purpose. The equivalent of the **distsum(D)** function is using the '**+=**' operator and adding the volume to variable.
distpercentbelow(D,f) **distpercentabove(D,f)**	These two functions tell you what percentage of entries are below or above a given number. Therefore it acts like an inverse of 'distcutoff'. It can be used to construct cumulative histograms. It returns a percent. It does not include 'f'. For example, **distpercentbelow(D,f) + distpercentabove(D,f)** ..will give a number close to 100% but not always exactly 100%, because values equal to 'f' (to 2 significant digits) will not be included in either side.
distsetaccuracy(n)	Sets the accuracy of distributions. The default is 2 significant digits (1%..10% errors). e.g. **at start** **declare let ignore = distsetaccuracy(3)** **end at** Applies to ALL distributions in the system.
	WARNING! This can cause memory usage to blow out!
distZscore(D,f)	Returns the Z-score of 'f' as measured by distribution 'D'.
distTscore(D,f)	Returns the t-stat of 'f' as measured by distribution 'D'.
ZscoreToPscore(f)	Converts a Z-score to a probability value, *on the assumption that the Z-score comes from a normally distributed distribution.* The value returned is a percent, representing the probability that the next number will be below 'f'. So if 'f' is near the bottom number in the distribution, then the return value 'p' will be close to zero. If 'f' is near the top number in the distribution, 'p' will be close to one.
PscoretoTscore(percent, number)	Converts a probability level to a students T stat
TscoreToPscore(number, number)	TscoretoPScore(tstat,N) converts a student t stat to a probability level.
TwoMeans(X,Y) **TwoMeans(X,Y,f)**	X and Y and distributions. This establishes whether or not there is a significant difference between the means for these two distributions. It returns a number corresponding to the "two mean test statistic 'r'". The second form, where a number 'f' is supplied, is used when you want to test the hypothesis that the means differ by 'f' ie. Mean(X) = Mean(Y) + f.
Correlation(X,Y,XY) **Correlation(X,Y,Sxy)**	This computes the Pearson's correlation coefficient 'r'. In the first form, you input distributions X, Y and XY where XY is the distribution of the product of the observations. In the second form you input 'Sxy' which is the sum of the product of the observations. You must compute the product of the observations yourself, rather than having this done automatically, because Alice has no feature of 'time-series', ie. the distributions X and Y on their own contain no information about which observations match with which. Giving it the distribution or sum of products provides the extra information required. Giving this function the sum Sxy is more efficient with memory, but giving it the distribution XY is conceptually easier and also incorporates an extra check that XY contains the correct number of observations. **Important: Correlation(X,Y,XY) will be *undefined* if:** **X is undefined or empty** **Y is undefined or empty** **XY is undefined or empty** **X contains a different number of observations to Y or XY.** **In other words, it is very important that you only insert values into any of the 3 distributions if both the observation 'x' and the observation 'y' are defined, for example:** **every 30 mins**

Function	Description
	if defined(spread / mpprice) then **let V <- volume(-30 mins)** **let S <- spread / mpprice** **let SV <- volume(-30 mins) * (spread / mpprice)** **end if** **end every** **at end** **print Correlation(S,V,SV)** **end at**
let X += Y	The 'let X += Y' statement allows you to merge the distribution 'Y' into 'X', i.e. combine all the observations from either distribution into one big distribution.

7.3 String functions

Function	Description
searchstr(A, b)	Returns the character position of the first occurrence of sub-string b in string A. It counts from 1, e.g. If 'b' occurs at the very beginning of 'A', then it will return '1'. (Note: prior to version 5, the numbering started from 0, which was inconsistent with 'substr' below).
strlen(string)	Returns the length of the string. 0 = empty string.
substr(S,a,b)	Returns the characters in position **a** to **b** from the sub-string **S**. For example: **substr("Hello Freddy", 7, 10)** will return **Fred** If you prefer to count the characters from the end of the string, make **a** and/or **b** negative. For example: **substr("Hello Freddy", -6, -10)** will return **llo F** In other words, each character in the string is numbered in 2 ways: 1 2 3 4 5 6 **7** **8** **9** **10** 11 12 H e l l o F r e d d y -12 -11 **-10** **-9** **-8** **-7** **-6** -5 -4 -3 -2 -1 **NOTE**: 1. Spaces are counted as one character as well. 2. *Alice* always identifies characters to be returned from the start of the sub-string, irrespective of the order in which the characters are identified. 3. If the sub-string you're after isn't in the string, eg. substr("XAO", 4, 4), substr will then return you an empty string.
strwildcardmatch(string1, string2)	Does string2 match with string1 if string1 is a wildcard string?
strtolower(string) **tolowecase(string)**	Function to convert a string to lower case string
strtoupper(string) **touppercase(string)**	Function to convert a string to upper case string
strtok(string1, string2)	Function to tokenize string1 with string2 delimiters
strieq(string1, string2)	Ignoring case, is string1 equal to string2?
date(<string>)	This converts the string into a date in dd/mm/yyyy or mm/dd/yyyy or yy-mm-dd format. The format in which the date is converted to depends on the date format selected in Preferences.
format(X,width, decimals)	This converts a numerical value (ie. **price**, **value**, **volume**, **number** and **percent**) to a string, without any formatting such as commas, prefixes, punctuation characters and/or exponential notation. You can specify the **width** and number of **decimal places**. If the **number** is too wide for **width**, then it will display the full **number** ie. it will exceed **width** in order to print a correct **number**. This means that if you want the **number** printed without leading spaces, set the **width** to zero. This function is particularly useful for getting numbers, which are generated through **print** *statements*, to line up in the output window. For example: **format(**$2,500.75**,10,4)** = 2500.7500_ _ **format(**x1,234,567**,10,0)** = 1234567_ _ _ _ **format(**-50%**,4,4)** = -50.0000 (If X is undefined, then it will return an empty string i.e. a zero length string).
format(<time>)	This formats the time so that it is always displayed as HH:MM:SS. Without this function, times will either be displayed as HH:MM:SS or HH:MM.
format(<string>, width)	This formats the string by removing all commas within the string.
security(string)	Converts a string representing a security, house etc. code into the entity itself.

Function	Description
house(string)	
trader(string)	
client(string)	
index(string)	

7.4 Current object functions

Function	Description

security	The current security.
	The security is always defined when used inside an **on …** *when clause*. In particular, it is defined as the security currently being evaluated.
	For example:
	`on trade`
	`  if volume > x1000000 then`
	`  print "Large trade in [security]"`
	`  end if`
	`end on`
	In the above **on trade** rule, the **[security]** printed will be the security in which the trade was executed in.
	
	The security however is not defined when used inside an **at …** or **every …** *when clause*. Thus depending on the rule being written, it may be necessary to use the **per security** statement within the **at …** or **every …** *when clause* to instruct *Alice* to loop through every security whilst evaluating the rule.
	For example:
	To give the new variable **cum_volume** which is indexed by security a starting volume, say
	`at start`
	`  declare cum_volume : volume`
	`end at`
	
	`at start`
	`  per security`
	`  let cum_volume[security] = x0`
	`  end per`
	`end at`
	
	To limit the securities over which *Alice* would loop through, you could also add an **if** or **with** *statement* following **per security**.
	For example:
	To assign different thresholds to different securities at the start of the day, say
	`at start`
	`  declare vol_threshold : volume`
	`end at`
	
	`at start`
	`  per security`
	`  if securityfield("TY") = "1" then`
	`  let vol_threshold[security] = x1000000`
	`  else`
	`  let vol_threshold[security] = x5000000`
	`  end if`
	`  end per`
	`end at`

Function	Description

house	The current house.
	Similarly to **security**, **house** is defined in
	a) **on …** *when clauses* relating to orders (eg: **on entord**, **on delord**); and
	b) **at …** and **every …** *when clauses*, if they are followed by a **per house** statement.
	For example:
	`at start`
	`  declare total_order : volume`
	`end at`
	
	`at start`
	`  per house`
	`  let total_order[house] = x0`
	`  end per`
	`end at`
	
	`on entord`
	`  let total_order[house] += volume`
	`end on`
	In the above example, the new variable **total_order** for every house is assigned an initial value of x0 volume at the start of the day. During the day after every entered order, the volume of the order is then added to the **total_order** variable of the house who entered the order.
	
	NOTE: Unlike security, **house** is never defined in a **on … *when clauses*** relating to trades (eg: **on trade**, **on mkttrade**). This is because a trade is made up of two sides which may or may not be the same house. In order to identify the houses responsible for the trade, the **buyerh** and **sellerh** functions should be used instead.

buyerh	The house on the buy side of the current trade.
	Only defined inside an **on …** when clauses relating to trades (eg: **on trade**, **on mkttrade**).
	For example:
	`on trade,entord`
	`  if volume > x1000000 and`
	`  change(trueprice,closeprice(-1)) <= 25% then`
	`  alert 102,"PRICE RISE","PRICE RISE :`
	`  [change(trueprice,closeprice(-1))], [buyerh] was buying from`
	`  [sellerh]",`
	`  house = buyerh`
	`  end if`
	`end on`

sellerh	The house on the sell side of the current trade.
	Only defined inside an **on …** when clauses relating to trades (eg: **on trade**, **on mkttrade**).
	For example:
	`on trade,entord`
	`  if volume > x1000000 and`
	`  change(trueprice,closeprice(-1)) >= -25% then`
	`  alert 101,"PRICE FALL","PRICE FALL :`
	`  [change(trueprice,closeprice(-1))], [sellerh] was selling to [buyerh]",`
	`  house = sellerh`
	`  end if`
	`end on`

Function	Description
trader *(may not be available)*	The current trader. **Trader** is defined in a) **on …** *when clauses* relating to orders (eg: **on entord**, **on delord**); and b) **at …** and **every …** *when clauses*, if they are followed by a **per trader** statement.
buyert *(may not be available)*	The trader on the buy side of the current trade. Only defined inside an **on …** when clauses relating to trades (eg: **on trade**, **on mkttrade**).
sellert *(may not be available)*	The trader on the sell side of the current trade. Only defined inside an **on …** when clauses relating to trades (eg: **on trade**, **on mkttrade**).
client *(may not be available)*	The current client. **Client** is defined in a) **on …** *when clauses* relating to orders (eg: **on entord**, **on delord**); and b) **at …** and **every …** *when clauses*, if they are followed by a **per client** statement.
buyerc *(may not be available)*	The client on the buy side of the current trade. Only defined inside an **on …** when clauses relating to trades (eg: **on trade**, **on mkttrade**).
sellerc *(may not be available)*	The client on the sell side of the current trade. Only defined inside an **on …** when clauses relating to trades (eg: **on trade**, **on mkttrade**).
time	The current time.
reporttime	The time the trade was reported to the market. Only defined inside an **on …** when clauses relating to trades (eg: **on trade**, **on mkttrade**). For regular on market trades reporttime will be the same as **time**, but may differ for off market trades.
executetime	The time the trade was executed. Only defined for trades and deltrades.
date	The current date.
alertdate	The date (or first date) on which the *Alice* program is being ran on. It would generally be used in the Benchmark Section to refer to the actual alert date as oppose to the dates used to calculate the benchmarks. NOTE: If the *Alice* program is being ran over multiple days, the **alertdate** function will return the first date in the date range. For example: **benchmarks_below : bentradingdays(3 days)** **at daystart** **print "Now doing [date], file is running on [alertdate]"** **end at** If the above *Alice* program is ran on the 16/9/98 - 18/9/98, the results from the print statement will be: Now doing 11/09/1998, file is running on **16/9/1998** Now doing 14/09/1998, file is running on **16/9/1998** Now doing 15/09/1998, file is running on **16/9/1998**
alertdate2	Similar to the **alertdate** function except that this function returns the second date in calibration run date range. This function is thus only useful when running alerts across a range of dates. It will always return the latter date in the date range. For example: **benchmarks_below : bentradingdays(3 days)** **at daystart** **print "Now doing [date], file is running on [alertdate] - [alertdate2]"** **end at** If the above *Alice* program is ran on the 16/9/98 - 18/9/98, the results from the print statement will be: Now doing 11/09/1998, file is running on 16/9/1998 **- 18/9/98** Now doing 14/09/1998, file is running on 16/9/1998 **- 18/9/98** Now doing 15/09/1998, file is running on 16/9/1998 **- 18/9/98**
day(date)	The day in the date specified. For example:

Function	Description
	day(30/6/1995**)** will return the number 30. **day(**30/6/1995 - 1**)** will return the number 29 derived from the 29/6/1995. **day(**30/6/1995 + 1**)** will return the number 1 derived from the 1/7/1995.
date(year,month,day)	This builds a date out of a year, month and day-of-month. For example : date (2000,12,25) = 25/12/2000
month(date)	The month in the date specified. For example: **month(**30/6/1995**)** will return the number 6. **month(**alertdate**)** will return 9 if the *Alice* program is being ran on the 18/9/1998.
year(date)	The year in the date specified. For example: **year(**30/6/1995**)** will return the number 1995.
publichol(date)	Returns a boolean indicating whether the date specified is a public holiday or not. **publichol(date) = true** : is a public holiday **publichol(date) = false** : is not a public holiday
dayofweek	The current day-of-the-week. **dayofweek = 0** : Monday, **dayofweek = 4** : Friday, **dayofweek = 1** : Tuesday, **dayofweek = 5** : Saturday, **dayofweek = 2** : Wednesday, **dayofweek = 6** : Sunday. **dayofweek = 3** : Thursday,
dayofweek(date)	Similar to the **dayofweek** function except that this function takes a date parameter, meaning it identifies the day-of-the week of the date specified. For example: **at start** **declare which_day : string** **declare d : date** **end at** **at start** **print "Public Holidays for the year [year(date)] are :"** **for d = [date] ; year(d) = year(date) ; d = d + 1 do** **if dayofweek(d) = 0 then** **let which_day = "Monday"** **elsif dayofweek(d) = 1 then** **let which_day = "Tuesday"** **elsif dayofweek(d) = 2 then** **let which_day = "Wednesday"** **elsif dayofweek(d) = 3 then** **let which_day = "Thursday"** **elsif dayofweek(d) = 4 then** **let which_day = "Friday"** **elsif dayofweek(d) = 5 then** **let which_day = "Saturday"** **elsif dayofweek(d) = 6 then** **let which_day = "Sunday"** **end if** **if publichol(d) = true then** **print " [which_day] the [d]"** **end if**

Function	Description
	end for end at If the above *Alice* program is ran on the 2/1/1998, the **for** *statement* will instruct *Alice* to scroll through each day in the year 1998 (starting from the 2/1/1998) and if the date is a public holiday, print the day of the week and the date of the public holiday.
mplid(security) **mplid(index)**	Returns the mplid of the security or index. This is helpful where a stock has been renamed or ceased trading and another security now trades under that name. Each security has a unique mplid which does not change. See tip under the read function.

7.5 Current transaction functions

Function	Description
board *(may not be available)*	Returns the board which the current transaction took place in. The actual string returned is the code of the flag which represents this board. Note that boards are represented by flags. Note also that flags typically have 2-letter codes, but can actually have any number of letters. If you want to match an ASTS trading system, boards are given 4-letter codes. Refer to Appendix A of your SMARTS User Manual for the list of available boards in your market. For example: **on trade** **if volume > x1000000 and change(trueprice,closeprice(-1)) > 20% then** **alert 101,"PRICE RISE","PRICE RISE : caused by trade which was executed in the [board] board."** **end if** **end on**
boardname *(may not be available)*	Returns the board long name which the current transaction took place in.
buy_or_sell	Returns the type of order. This value is only defined when used with the **on …** *when clause* relating to orders (**on entord**, **on delord, on amdord**) or during a **per order** clause. The list of order types are **buy_or_sell = "B"** : buy order (bid) **buy_or_sell = "S"** : ask order (sell) For example: To alert on bid orders which result in a price change between the current trueprice and the previous close price to be greater than 20% say **on entord** **if buy_or_sell = "B" and** **change(trueprice,closeprice(-1)) > 20% then** **alert 102 …** **end if** **end on** ** See also "is_ask" and "is_bid" functions later in this section.*

Function	Description
childmkt	This only applies to sites that have a 'meta-market' set up. It returns a string telling you what constituent market the current security belongs to.
controlstatus **controlstatus(security)**	Returns the new market/security status. Is only defined when used with the **on control** *when clause*. The list of control status are **controlstatus = "N"** ≈ CONTL = OPEN **controlstatus = "G"** ≈ CONTL = OPENING **controlstatus = "C"** ≈ CONTL = CLOSED **controlstatus = "S"** ≈ CONTL = SUSPEND **controlstatus = "E"** ≈ CONTL = PRE OPEN **controlstatus = "H"** ≈ CONTL = TRADING HALT Other values are possible as each site may have configurable control statuses. Please refer to site specific documentation or contact your local SMARTS administrator for more information on any configured control statuses at your site. Giving controlstatus an argument in the form of a security will obtain the controlstatus of that security at the current time. For example: **on control** **if controlstatus = "S" then** **alert 740,"SUSPENDED","[security] was suspended at [time]",** **reissue = "740S"** **end if** **end on**

InControlGroup	Is the specified security in the group of the current CONTROL (i.e. status-change) message?
	This function is only defined when used with the **on control** *when clause.*
	For example:
	`on control`
	`with ^STEL`
	`if InControlGroup(security) then`
	`print "Security TLKM is in the current control group : [ControlStatus]"`
	`end if`
	`end with`
	`end on`

infofield	Extracts the field tag as a string, this function is only defined in the **on info** *when clause.*
infofields	Returns the string of all info fields (except the full text field) strung together, this function is only defined in the **on info** *when clause.*
infotext	Returns the full text of the information announcement, this function is only defined in the **on info** *when clause.*
infotype	Returns the type of the information announcement, this function is only defined in the **on info** *when clause.*
infosensitivity	Is the information announcement price sensitive (positive, negative, neutral or undecided)? This function is only defined in the **on info** *when clause.*
Infotranstype	Returns a string to indicate whether the announcement is a new announcement, an amendment or deletion. Possible values "AnnfoNew", "AnnfoAmend", and "AnnfoDelete".

is_ask	Returns the sell side of the order. This value is only defined when used with the **on …** *when clause* relating to orders (**on entord**, **on delord, on amdord**) and during a **per order** clause
	For example:
	To alert on sell orders which result in a price change between the current trueprice and the previous close price to be greater than 20% say
	`on entord`
	`per order`
	`if is_sell and`
	`change(trueprice,closeprice(-1)) > 20% then`
	`alert 102 …`
	`end if`
	`end per`
	`end on`
	This function supersedes the **buy_or_sell** one and saves users from having to declare the function to take on a "B" or "S" value.

is_bid	Returns the buy side of the order. This value is only defined when used with the **on …** *when clause* relating to orders (**on entord**, **on delord, on amdord**) and during a **per order** clause
	For example:
	To alert on bid orders which result in a price change between the current trueprice and the previous close price to be greater than 20% say
	`on entord`
	`per order`
	`if is_bid and`
	`change(trueprice,closeprice(-1)) > 20% then`
	`alert 102 …`
	`end if`
	`end per`
	`end on`

	This function supersedes the **buy_or_sell** one and saves users from having to declare the function to take on a “B” or “S” value.

Oldorderid	The previous ID of an order in the case of an **on amdord** transaction.

oldprice	The previous price of an order in the case of an **on amdord** transaction.
	For example:
	`on amdord`
	`  if change(price,`**`oldprice`**`) > 0% and`
	`  change(price,`**`oldprice`**`) > 3 * tickup(`**`oldprice`**`) then`
	`  alert 103,"LARGE PRICE AMEND","…`
	`  end if`
	`end on`

oldvalue	The previous value of an order in the case of an **on amdord** transaction.
oldvolume	The previous volume of an order in the case of an **on amdord** transaction.
orderid	The ID of the current order. This function is only defined inside **on …** *when clauses* relating to orders (eg: **on entord**, **on delord**, **on amdord**).
orderidask	The ID of the ask order in a trade. This function is only defined inside **on …** *when clauses* relating to trades (eg: **on trade**, **on mkttrade**).
orderidbid	The ID of the bid order in a trade. This function is only defined inside **on …** *when clauses* relating to trades (eg: **on trade**, **on mkttrade**).
price	The price of the current transaction.
quoteask **quotebid** *(only applicable to quote driven markets)*	The price of the ask (or bid) side of the quote. This function is only defined inside an **on quote** *when clause*.
quotevolask **quotevolbid** *(only applicable to quote driven markets)*	The volume attached to the quote. This function is only defined inside an **on quote** *when clause*.
tradeid	The ID of the current trade. Can only be used inside an **on …** *when clauses* relating to trades (eg: **on trade**, **on mkttrade**).

transtype	Returns a string representing the current transaction type.
	The list of transaction types available will usually include the following: **ENTER**, **AMEND, DELET**, **TRADE**, **CANTR**, **CONTL** or **ANNFO**. This list however may vary between markets.
	For example:
	To cumulate and print the number of bid and sell orders entered, amended or deleted in each of the top 10 securities during the day, say
	`at start`
	`  declare num_trans : number`
	`end at`
	`on entord, amdord, delord`
	`  let num_trans[security,`**`transtype`**`,buy_or_sell] += 1`
	`end on`
	`at dayend`
	`  per security`
	`  if rank(date -7 days) <= 10 then`
	`  print "[security]`
	`  [num_trans[security,`**`"ENTER"`**`,"B"]]`
	`  [num_trans[security,`**`"DELET"`**`,"B"]]`
	`  [num_trans[security,`**`"AMEND"`**`,"B"]]`
	`  [num_trans[security,`**`"ENTER"`**`,"S"]]`
	`  [num_trans[security,`**`"DELET"`**`,"S"]]`
	`  [num_trans[security,`**`"ANNFO"`**`,"S"]]`
	`  end if`
	`  end per`
	`end at`

uvolume	The hidden volume on an order.
value	The dollar value of the current transaction (includes only on market trades). For an AMEND transaction (*known as REDUCE in SEHK), this returns the *new dollar value* of the order.
volume	For a TRADE: the number of units transacted. For an ENTORD/DELORD/per order: the number of units on the order (including hidden volume). For an AMEND transaction (*known as REDUCE in SEHK), this returns the *new volume* of the order.

7.6 Query functions

Query functions which take a time-parameter or date parameter recognises both absolute time/date parameters and time/date-difference parameters. For example:

Absolute time	**volume(11:00)**	this function will give you the volume since 11:00 o-clock in the morning.
Absolute date	**volume(15/9/98)**	if the current date is the 18/9/98, this function will give you the total volume between the 15/9/98 - 18/9/98.
Time-difference	**volume(time - 1:00)**	this function will give you the volume traded over the past hour prior to the current time. This function can also be written as **volume(-1:00)** **volume(-1 hours)**
	volume(- 30 minutes)	this function will give you the volume traded over the previous 30 minutes.
Date-difference	**volume(date - 10 days)**	if the current date is the 18/9/98, this function will give you the total volume traded between the 8/9/98 - 18/9/98.
	volume(trday(-10))	if the current date is the 18/9/98, this function will give you the total volume traded between the 4/9/9/8 - 18/9/98.

By default, the following query functions will refer to the current security/house/trader/client. There are however some functions which allows the user to explicitly specify a security/house/trader/client. For example:

holding(security,house)	this function can take a security and a house parameter.
cash(house)	this function can only take a house parameter.

7.6.1 Date Query Functions

Function	Description

trday(number)
trday(date,number)
	Adds/subtracts the specified number of trading days to/from the current date (or specified date). The number specified should be negative if you want to go backwards in time.
	NOTE: This function excludes the current date or the date specified from the number of days required.
	For example:
	trday(-5) if the current date is the 18/09/1998, this function will return the 11/09/1998.
	trday(5/9/98,-5) this function will return the 31/08/1998.
	volumeall(**trday(-5)**) if the current date is the 18/9/98, this function will return the total volume traded between the 11/9/98 - 18/9/98.
	To compare the change in trading volume done so far today with the average trading volume over the previous 10 trading days, say
	`on trade`
	`if change(volumeall(0:00:00),`
	`(volumeall(trday(-10)) - volumeall(0:00:00)) ... then`
	`alert ...`
	`end if`
	`end on`

ntrdays(date)
ntrdays(date,date)
	Gives you the number of trading days between the current date and the specified date, or between the two specified dates.
	NOTE: This function includes the current date or the dates specified when counting the number of trading days.
	For example,
	ntrdays(31/8/1998) if the current date was the 18/9/98, this function will return 6.
	NOTE: This function counts the current date as one day as well. Thus,
	ntrdays(18/9/98,18/9/98)
	will return 1.
	ntrdays(10/4/98, 15/4/98) if the 10/4/98, 12/4/98 and 13/4/98 are public holidays then this function will return 2.
	To cumulate the volume traded by each house 5 trading days prior to the alert date in the benchmark section, say
	`Benchmarks_below : 30 days`
	`on trade`
	`if ntrdays(date,alertdate) > 6 then`
	`let vol_5days[buyerh] += volume`
	`let vol_5days[sellerh] += volume`
	`end if`
	`end on`
	NOTE: A different method of getting the same results as above is shown below in the example relating to the **bentradingdays(number)** function

Function	Description
trdaydiff(date1,date2)	This function returns the number of trading days in between the current date and specified date. For example, if date1 = date2, it returns 0. If date 1< date 2, it returns the number of trading days in between (including date1 but excluding date2). If date1 > date2 then it returns a negative value. **NOTE** : This function supersedes the ntrdays function.

bentradingdays(number)	This function is provided specifically to help you specify a number of trading days rather than calendar days in the "benchmarks_below" section.
	NOTE: This function excludes the alert date from the number of trading days required.
	For example,
	To generate benchmarks from 30 trading days prior to the alert date, in the benchmark section say:
	`benchmarks_below :` **`bentradingdays(30 days)`**
	
	To cumulate the volume traded by each house 5 trading days prior to the alert date in the benchmark section, say
	`benchmarks_below : 30 days`
	
	`on trade`
	`if date > alertdate –` **`bentradingdays(5 days)`** `then`
	`let vol_5days[buyerh] += volume`
	`let vol_5days[sellerh] += volume`
	`end if`
	`end on`
	
	NOTE: A different method of getting the same results as above was previously shown in the example relating to the **ntrdays(date)** function.

systemdate	Returns the date the Alice program is being run on.

7.6.2 Price Query Functions

Function	Description

ask
ask(time)
ask(board)
bid
bid(time)
bid(board)
	Returns the best bid/ask (ie price of the lowest ask order or highest bid order) in the current security at the current time or at the time specified.
	Warning: if you are currently in an ***on entord*** clause, then the bid will have already been updated with the current order if it is higher than the previous bid. To get the best bid price before the current transaction, use the when-clause: ***before entord*** instead, or use the functions **bidbefore/bidafter** and **askbefore/askafter**. The same applies to all transactions.
	NOTE: For efficiency reasons, the (time) functions are only correct to the nearest 5 minutes (or can be configured to be less). To identify the exact time intervals used by the **ask(time)** or **bid(time)** function, use
	sampletime0(time) to identify the start of the time interval used
	sampletime1(time) to identify the end of the time interval used
	For example:
	To print the best ask and bid at market close (ie 16:00) and the best ask and bid 15 minutes prior to market close, say:
	`at 16:00`
	`per security`
	`print "[security] [ask] - [bid] at [time]`
	`[ask(time-00:15)] - [bid(time-00:15)] at [time-00:15]"`
	`end per`
	`end at`
	Also returns the best bid/ask price for the current security, current time and for a specified board. You may give the functions the board code, board short name or board long name.
	For example :
	`at 11:00`
	`per security`
	`print "Best bid for [security] on the Regular Board is [bid("RG")]"`
	`print "Best bid for [security] on the Immediate Board is [bid("Imme")]"`
	`end per`
	`end at`

askbefore **bidbefore**	Suppose an ENTER or AMEND transaction causes a trade or sequence of trades. If we are processing an 'on entord' or 'on trade' clause, and we call the 'bid' or 'ask' functions, then we will see an overlapping market (i.e. bid >= ask). But suppose we want to know what the spread was immediately before the order. In this case we use these functions. These functions return the spread (i.e. best bid/ask) as at the last time where there was a non-overlapping market. This normally equals the spread 1 second before the current transaction. On the last trade of the sequence, i.e. the trade which clears the market, these functions will return the previous best bid/ask, so you can compare the previous non-overlapping spread with the current non-overlapping spread. However any subsequent transactions will then put the current non-overlapping spread into these values. These functions are not used so much anymore now that we have the **before entord / before trade** etc. when-clauses.
askafter **bidafter** (also called **clearedask**/ **clearedbid**)	Returns the best ask/bid price, after simulating matching on any overlapping orders during the auction period.
clearedprice	Returns the clearing price during the auction(pre-open) period or the trueprice at other

Function	Description
clearedprice(time)	times.
clearedvol	Returns the number of shares traded during a simulation of the matching algorithm on the current snapshot of the orderbook.
spread **spread(time)**	Returns the spread of the current security at the current time or time specified. Spread is the difference between the best ask and best bid at the current time or time specified. ie **spread** = ask – bid **spread(time)** = ask(time) - bid(time)
price(time)	Equivalent to **trueprice(time)**.
nominalprice	Returns the nominal price (face value) of a debt instrument
openprice(date) **openprice(number)**	Returns the first traded price at the start of trading on the date specified or identified. Excludes off-market trades and odd-lot trades.
closeprice(date) **closeprice(number)**	Returns the last traded price at the close of trading on the date specified or identified. Excludes off-market trades and odd-lot trades.
IEP	Returns the indicative equilibrium price
indexvalAverage(index, date)	The average value for this index on the date specified
indexvalClose(index, date)	The last value for this index on the date specified
indexvalMax(index, date)	The maximum value for this index on the date specified
indexvalMin(index, date)	The minimum value for this index on the date specified
indexvalOpen(index, date)	The first value for this index on the date specified
lastprice	Returns the previous trade price on the current date or previous date with trades. Excludes off-market trades and odd-lot trades.
maxprice (date) **maxprice(number)**	Returns the highest trade price on the current date or date identified. Excludes off-market trades and odd-lot trades.
minprice(date) **minprice(number)**	Returns the lowest trade price on the current date or date identified. Excludes off-market trades and odd-lot trades.
mpprice **mpprice(time)**	Returns the midpoint price of the current security at the current time or time specified. Midpoint price is equivalent to the sum of the best ask and bid divided by two. ie: **mpprice** = (ask + bid) / 2 **mpprice(time)** = (ask(time) + bid(time)) / 2
trueprice **trueprice(time)**	Returns the 'true price' of a security at the current time or time specified. 'True price' is defined as the last-traded-price (excluding off-market trades), unless the last-traded-price has been overtaken by a bid or ask. It was designed in order to improve on the 'lastprice' and 'mpprice' prices. See Appendix B of the SMARTS User Manual for a full definition of true price.
tickup(price) **tickdown(price)**	Returns the price-step (or minimum price-tick) for the price specified. 'tickup' is the price-step above the price specified, and 'tickdown' is the price-step below the price specified. The distinction only matters at prices on the border between two price steps (or price ticks). For example: Say a market has the following price ticks : $1.000 - $4.995 tick = $0.005 $5.000 - $9.99 tick = $0.01 **tickup(**$6.05**)** = $0.01 **tickdown(**$6.05**)** = $0.01 **tickup(**$5.00**)** = $0.01 **tickdown(**$5.00**)** = $0.0005 **tickup(**$4.995**)** = $0.005 **tickdown(**$4.995**)** = $0.0005 **tickup(**$4.005**)** = $0.005 **tickdown(**$4.005**)** = $0.0005

Function	Description
vwap **vwap(date)** **vwap(time)**	The volume-weighted average trade price as at the current date or on the date specified (includes on market trades only). Returns a price. $vwap = \frac{(price\ x\ vol\ trade\ 1) + (price\ x\ vol\ trade\ 2) + ... + (price\ x\ vol\ last\ trade)}{total\ volume}$ For example: If the following two trades were executed in security x on the current date, the following alice rule using the **vwap** function will return: Trade 1 : 100 units @ $1.00 Trade 2 : 100 units @ $2.00 **at dayend** **per security** **print "VWAP of [security] on [date] = [vwap]."** **end per** **end at** Output : **VWAP of <security x> on <current date> = $1.50** where $vwap = \frac{\$1.00\ x\ 100 \quad + \quad \$2.00\ x\ 100}{200} = \frac{\$300}{200} = \$1.50$ **NOTE**: If **vwap** is evaluated after every trade, **vwap** will represent the volume weighted trade price at that current time on the current date. If **vwap** is evaluated at a specific time using the (time) command, it will return the volume weighted trade price at the current time since the beginning of trading. If **vwap** is evaulated at regular intervals, **vwap** will represent the volume weighted trade price of the interval since the beginning of trading.

7.6.3 Value Query Functions

Function	Description
value(time) **value(date)** **value(date1,date2)** **cancelled_value(time)**	The dollar value turnover of ON MARKET trades only in the current security since a given time or date. Note, the definition of on market trade here is: any trade that follows the vwap_trades config. There are 4 forms of this function: • If you specify a *time parameter*, the **value(time)** function will give you the value of all on market trades on the current day from the specified time up to the current time. For example: **value(12:00)** this will give you the total value of trades from 12:00 till the current time. **value(-1:00)** this will give you the total value of trades over the past hour. **value(-5 minutes)** this will give you the total value of trades over the past 5 minutes. **NOTE**: For efficiency reasons, this function is only correct to the nearest 5 minutes (or can be configured to be less). To identify the exact time intervals used by the **value(time)** function, use sampletime0(time) to identify the start of the time interval used sampletime1(time) to identify the end of the time interval used • **value(date)** will give you the value of trades from and including the specified date up to the current time on the current date. • **value(date1,date2)** function will give you the value of trades from date1 to date2 inclusive. Does not include the current date or any date beyond the current date. • **cancelled_value(time)** function will give you the value of all cancelled on market trades on the current day from the specified time up to the current time.
valueofficial(time) **valueofficial(date)** **valueofficial(date1,date2)** **cancelled_valueofficial(time)**	The total dollar value of all OFFICIAL trades in the current security since this time or date. **NOTE**: For efficiency reasons, this function is only correct to the nearest 5 minutes (or can be configured to be less). To identify the exact time intervals used by the **valueofficial(time)** function, use sampletime0(time) to identify the start of the time interval used sampletime1(time) to identify the end of the time interval used The definitions of each of these functions are similar to **value(time)**, **value(date)**, **value(date1,date2)** and **cancelled_value(time)** except that they look at official trades, i.e. trades that follow either the vwap_trades config or the total_adjustment_trades config.
valueall(time) **valueall(date)** **valueall(date1,date2)**	The total dollar value of all ON MARKET and OFF MARKET trades in the current security since this time or date. **NOTE**: For efficiency reasons, this function is only correct to the nearest 5 minutes (or can be configured to be less). To identify the exact time intervals used by the **valueall(time)** function, use sampletime0(time) to identify the start of the time interval used sampletime1(time) to identify the end of the time interval used The definitions of each of these functions are similar to **value(time)**, **value(date)** and **value(date1,date2)** except that they look at both on and off market trades.
Cash **cash(house)** *(only available in markets with cash position*	Returns the amount of cash held by the current house or specified house/s **as at** the start of the day. It has two forms, • one which takes the current house and needs no parameters. For example:

Function	Description
information)	**cash** this will return the amount of cash held by the current house and security as at the start of the day. • one which works with parameters**: (house).** For example: **cash(@B287)** this will return the amount of cash held by B287 since the start of the day.
marketcap	Market capitalisation, i.e. issuedunits * trueprice

7.6.4 Volume Query Functions

Function	Description
volume(time) **volume(date)** **volume(date1,date2)** **cancelled_volume(time)**	The number of units of ON MARKET trades only traded in the current security since a given time or date. Note, the definition of on market trade here is: any trade that follows the vwap_trades config. The definitions of each of these functions are similar to **value(time)**, **value(date)** and **value(number)** except that they return a volume. **NOTE**: For efficiency reasons, this function is only correct to the nearest 5 minutes (or can be configured to be less). To identify the exact time intervals used by the **volume(time)** function, use sampletime0(time) to identify the start of the time interval used sampletime1(time) to identify the end of the time interval used • **volume(date)** will give you the value of trades from and including the specified date up to the current time on the current date. • **volume(date1,date2)** function will give you the value of trades from date1 to date2 inclusive. Does not include the current date or any date beyond the current date. • **cancelled_volume(time)** function will give you the value of all cancelled on market trades on the current day from the specified time up to the current time
volumeofficial(time) **volumeofficial(date)** **volumeofficial(date1,date2)** **cancelled_volumeofficial(time)**	The total number of units traded in all OFFICIAL trades in the current security since this time or date. **NOTE**: For efficiency reasons, this function is only correct to the nearest 5 minutes (or can be configured to be less). To identify the exact time intervals used by the **volumeofficial(time)** function, use sampletime0(time) to identify the start of the time interval used sampletime1(time) to identify the end of the time interval used The definitions of each of these functions are similar to **value(time)**, **value(date)**, **value(date1,date2)** and **cancelled_value(time)** except that they return a volume and that they look at official trades, i.e. trades that follow either the vwap_trades config or the total_adjustment_trades config.
volumeall(time) **volumeall(date)** **volumeall(date1,date2)**	The total number of units traded in all ON MARKET and OFF MARKET trades in the current security since this time or date. **NOTE**: For efficiency reasons, this function is only correct to the nearest 5 minutes (or can be configured to be less). To identify the exact time intervals used by the **volumeall(time)** function, use sampletime0(time) to identify the start of the time interval used sampletime1(time) to identify the end of the time interval used The definitions of each of these functions are similar to **value(time)**, **value(date)** and **value(date1,date2)** except that they return a volume and they look at both on and off market trades.
holding	Returns the volume of shares held by the current house in the current security or

Function	Description
holding(security, house) *(only available in markets with holdings information)*	specified house/s in the specified security/ies **as at** the start of the day. It has two forms, • one which takes the current house and security and needs no parameters. For example: **holding** this will return the volume of shares held by the current house and security as at the start of the day. • one which works with parameters: **(house,security)**. For example: **holding(@B287,^CBTP)** this will return the volume of shares held by B287 in CBTP since the start of the day.

findordervol(orderid)	Gives you the volume remaining on an order, identified by orderid.
	It is intended mainly for use as follows:
	`on trade`
	`... findordervol(orderidbid) ...`
	`... findordervol(orderidask) ...`
	`end on`
	Note: (a) if you use this inside an "on trade", it gives you the volume _after_ the trade, and therefore returns zero if orderidbid or orderidask is fully traded off;
	(b) it can be used in other contexts, but only if the orderid belongs to the current security and it returns undefined if the order is fully traded off.

issuedunits	The number of shares on issue
bidvolatstep	Gives you the bid volume at the specified orderbook price step. This is the equivalent of using the trades screen in Replay with the orders grouped at each price step.
askvolatstep	Gives you the ask volume at the specified orderbook price step. This is the equivalent of using the trades screen in Replay with the orders grouped at each price step.
bidvolatdepth	Gives you the bid volume at the specified orderbook depth. This is the equivalent of using the trades screen in Replay with the orders separated at each price step.
askvolatdepth	Gives you the ask volume at the specified orderbook depth. This is the equivalent of using the trades screen in Replay with the orders separated at each price step.

7.6.5 Trade Count Query Functions

Function	Description
tcount(time) **tcount(date)** **tcount(date1,date2)** **cancelled_tcount(time)**	The number of ON MARKET trades only executed in the current security since a given time. Note, the definition of on market trade here is: any trade that follows the vwap_trades config. The definitions of each of these functions are similar to **value(time)**, **value(date)**, **value(date1,date2)** and **cancelled_value(time)** except that they return a number.
tcountofficial(time) **tcountofficial(date)** **tcountofficial(date1,date2)** **cancelled_tcountofficial(time)**	The total number of all OFFICIAL trades in the current security since this time or date. The definitions of each of these functions are similar to **value(time)**, **value(date)**, **value(date1,date2)** and **cancelled_value(time)** except that they return a number and they look at at official trades, i.e. trades that follow either the vwap_trades config or the total_adjustment_trades config.
tcountall(time) **tcountall(date)** **tcountall(date1,date2)**	The total number of all ON MARKET and OFF MARKET trades in the current security since this time or date. The definitions of each of these functions are similar to **value(time)**, **value(date)** and **value(date1,date2)** except that they return a number and they look at both on and off market trades.

7.6.6 Miscellaneous Query Functions

Function	Description
accruedint	Returns the amount of interest accrued on this bill.

Function	Description
clientfilter(string, client)	Returns true if the string given is a valid filter for the given client. Similar to client application client "new set" filtering functionality. For example: per client if (clientfilter("HO=33", client) then print "[client] is part of house 33" end if end per
controlstatus(security)	Returns the status of the specified security. The return value is identical to that of **controlstatus** without parameters (see section 7.5 for a description of the **controlstatus** function). The list of valid **controlstatus(security)** strings are **controlstatus(security) = "N"** ≈ CONTL = OPEN **controlstatus(security) = "G"** ≈ CONTL = OPENING **controlstatus(security) = "C"** ≈ CONTL = CLOSED **controlstatus(security) = "S"** ≈ CONTL = SUSPEND **controlstatus(security) = "E"** ≈ CONTL = PRE OPEN For example To determine which securities are still suspended at 3:00 pm, type: **at 15:00** **per security** **if controlstatus(security) = "S" then** **alert 701,"STILL SUSPDENDED","[security] is still suspended at [time]",** **reissue = "701S"** **end if** **end per** **end on**
dilution(date)	Returns the dilution factor since the given date. For example: a) If a security had a 2 for 1 split on the 31/1/1997, the dilution factor on any date prior to the 31/1/1997 (eg: **dilution(30/1/1997)**) will return the value 2. The value 2 means that prices prior to the 31/1/1997 were 2 times the prices on and after the 31/1/1997. To adjust previous prices to today's prices, divide all previous prices by the dilution factor of 2. b) If a dividend of a $1 was given on the 30/6/1997 and the security was trading around $5 before the dividend, the dilution factor on any date prior to the 30/6/1997 (eg: **dilution(1/6/1997)**) will return the value 1.25. The value 1.25 means that prices prior to the 30/6/1997 were 1.25 times the prices on and after the 30/6/1997. To adjust previous prices to today's prices, divide all previous prices by the dilution factor of 1.25.
dummynumber **dummydistribution** **dummypercent** **dummyboolean** **dummydate** **dummytime** **dummysecurity** **dummyhouse**	These functions return undefined values of the relevant type. There is one of these for each of the 15 Alice types. There are various situations where you might want an 'undefined' value as opposed to a zero value (or empty string etc.) One such situation is in order to free memory: whenever you assign an undefined value to a variable, the memory used by that variable is freed. This is particularly useful where *distribution* variables are used – because distributions use so much memory. Also, multidimensional arrays can use a lot of memory. However, now Alice has a special statement: **free**, for freeing an entire array in one go.

Function	Description
dummytrader **dummyclient** **dummyindex** **dummystring** **dummyvalue** **dummyprice** **dummyvolume**	**NOTE** : If a variable is declared with the '**zeroed**' syntax, then assigning an undefined value will delete it from memory, and if you query it again it will appear as zero.
frompot	The source settlement account.
topot	The destination settlement account.
herald(tag,security) **herald(tag,security,house)**	The herald database allows you to store numeric values attached to arbitrary securities/houses/traders/indexes and combinations of securities with (houses/traders/indexes). This function looks up field 'tag' in the herald database for this security or (security,house) and returns it as a number. To convert it to a volume, multiply by x1; to convert to a value, multiply by $1; to convert to a price, multiply by $1.00 and to convert to a percent, multiply by 100% (putting the 100% on the right hand side). For example: **print herald("weight",security,index) * 100%**
housefilter	Returns true if the string given is a valid filter for the given house. Similar to client application house "new set" filtering functionality. For example: per house if (housefilter("SC=BH", house) then print "[house] has BH in SC value" end if end per
indexval	'indexval' on its own with no parameters is used to get the index value of the current "on index" message.
indexval(X) **indexval(X,time)**	The value of the market indicator X, at the current time or at the specified time. For example: To print the value of the index every 30 minutes and the change since the start of the day, type: every 30 minutes per index print "Value of the DOWPEARCY index at [time] is **[indexval(X:DOWPEARCY)]**, at start was **[indexval(X:DOWPEARCY,10:01)]**, change = [change(**indexval(X:DOWPEARCY)**, **indexval(X:DOWPEARCY,10:01)**)]" end per end every
interestrate **interestrate(date)**	The risk free interest rate on the current date interestrate(date): the risk free interest rate on the specified date.
Ismarketmaker(security) **Ismarketmaker(house, security)** *(only applicable to certain markets)*	This function is available only in markets which have information about market-makers. Returns true or false depending on whether the current house or specified house is a market-maker in the specified security. For example: **ismarketmaker(@B287,^CBTP)**
Ismarketmakerh(security) **Ismarketmakerh(house, security)**	This function is available only in markets which have information about market-makers. Returns true or false depending on whether the current house or specified house is a

Function	Description
	market-maker in the specified security. For example: with @B287 **ismarketmakerh(^CBTP)** **end with**
Ismarketmakert(security) **Ismarketmaker(trader, security)**	This function is available only in markets which have information about market-makers. Returns true or false depending on whether the current trader or specified trader is a market-maker in the specified security. For example: with &**8015** **ismarketmakert(^CBTP)** **end with**
Isregisteredtrader(trader) **Isregisteredtrader(trader,security)** *(only applicable to certain markets)*	Returns true or false depending on whether the specified trader is a 'registered trader' in the specified security (or current security). For example: **isregisteredtrader(&8015,^0005)**
Istrading(date)	Returns a boolean (ie **true** or **false**) depending on whether the specified date is a trading day or not. For example: **istrading(1/1/1997)** will return **true** **istrading(4/1/1997)** will return **false**
memorystats	Prints statistics about memory usage: which arrays are using the most memory. This is useful if you want to optimise your Alice program or you are running out of memory. For example: **print memorystats** A table of the arrays consuming most memory will be printed to the screen (or 'stdout'). The table has columns for (a) the number of tuples i.e. entries, and (b) the number of bytes. For strings and distributions, a '+' symbol is displayed after the number of bytes to indicate that we don't know the exact number of bytes but it is at least this number. Arrays with less than 50 entries will not be printed.
namefieldmatch(string entity, string code, string field, date_id date, string value)	Does the given entity (security, house, trader, client, index, issuer, etc) indicated by the entity code, have the specified field in the date given. This function works with multivalue name field. For example: **namefieldmatch("security", "CBTP", 20/12/2003, "TY", "warrant")** **namefieldmatch("security", "SWIFT", 1/12/2004, "B1", "1")** **namefieldmatch("trader", "20-452", 20/2/2002, "HO", "20")**
oyield(price)	Returns the official yield on a price of P.
rank(date)	Returns the 'rank' of the current security. The rank of a security is determined by its total turnover value of all on and off market trades from the specified date up to but not including the current date. The security with the highest turnover over the period is given the rank 1 and securities which do not trade at all, are all given an equal rank of 99999. For example: To only evaluate the alert rule on the top 50 stocks ranked on their turnover over the past 30 days, type: **on trade** **if rank(alertdate - 30 days) >= 50 then** **if …** **alert ….** **end if**

Function	Description
	end if end on Note that you are not allowed to query the rank of just the current date, because in realtime mode this information is not available until the end of the day.
rank(date1,date1)	Similar to the two rank functions above. This function however returns the rank of a security based on the turnover value of all on and off market trades between the two dates specified. The turnover value on the two dates specified are also included unless it is the current alertdate.
securityfilter(string, security)	Returns true if the string given is a valid filter for the given security. Similar to client application security "new set" filtering functionality. For example: per security if (securityfilter("GN=4 and TY=equity", security) then print "[security] is in group 4 and is of type equity" end if end per
seconds(time)	Obtains the seconds part of a specified time
threshold **thishold**	Suppose the most recently executed comparison was "X > Y". Then 'threshold' will return 'X', and 'thishold' will return 'Y'. These functions are used internally to compute the 'intensity' parameter (unless you specify your own 'intensity' formula in an alert statement).
traderfilter(string, trader)	Returns true if the string given is a valid filter for the given trader. Similar to client application trader "new set" filtering functionality. For example: per trader if (traderfilter("HO=20", trader) then print "[trader] is from HO 20" end if end per
usercode	Obtains the user code of the current Alice user.
weighting_in_index(security) **weighting_in_index(index)**	What percentage does the current security hold in the specified index? Or what percentage does the current security hold in the specified index?

Function	Description
wasalertissued	Was an alert issued by the last alert command? Returns 0 for no, 1 for yes (new), and 2 for reissue.

flags ( )	Returns a boolean (ie **true** or **false**) depending on whether the current order or trade matches the specified transaction flags. Orders and/or trades which do not match the flags specified will be filtered out.
	For example:
	To print the details of all trades in a particular security within a specified time range plus a "Y" tag next to all trades which are short sale trades, type:
	
	`on trade`
	`  if security = ^CBTP then`
	`  if time >= 11:00 and time <= 12:00 then`
	`  if **flags(+SH)** then`
	`  let isshortsale = "Y"`
	`  else`
	`  let isshortsale = ""`
	`  end if`
	
	`  print "[time] [security], [volume] @ [price] ([value]),`
	`  [isshortsale]"`
	`  end if`
	`  end if`
	`end on`
	
	The syntax of the string inside the flags filter is the same as you see inside Spread, and the same used in the Facts language. You type a sequence of terms such as the following:
	+SH : Only trades with the "SH" flag will match
	-SH : Only trades *without* the "SH" flag will match
	Src=Internet : The 'Little fields' can also be used in the filter
	Src!=Internet : This example selects trades where the little-field "src" does not equal "Internet"
	-OF-od : Exclude all OF (off-market) trades and od (odd-lot) trades
	
	You can use 'and's and 'or's and 'not's inside this string, e.g.:
	flags(+SH and Src=Internet)
	
	however you might alternatively prefer to use logical operators outside the string, e.g.:
	flags(+SH) and flags(Src=Internet).
	
	NOTE:
	a) This function cannot be used to return the trade flags relating to a particular trade. This function can only confirm whether or not a trade contains a prespecified set of trade flags.
	b) The string of flags entered must not contain quote-marks.
	c) Each flag entered must be preceded by a '+' or '-'.
	d) Each little-field must be of the form: "<field>=<value>" or "<field>!=<value>"
	e) The full list of all the 2-letter flag codes and little fields and their descriptions can be found in Appendix A of the SMARTS User Manual.

7.7 Participant Identification Functions

Function	Description
house	The current house (see section 7.4 "Current Object Functions").
trader *(may not be available)*	The current trader (see section 7.4 "Current Object Functions").
client *(may not be available)*	The current client (see section 7.4 "Current Object Functions").
buyerh	The buying house during an on trade rule (see section 7.4 "Current Object Functions")..
sellerh	The selling house during an on trade rule (see section 7.4 "Current Object Functions")..
initiatorh	The house who initiated the current trade. Only defined inside an **on …** when clauses relating to trades (eg: **on trade**, **on mkttrade**). For example: To identify the house who initiated the trade which gave rise to the alert, type: **on trade** **if …. then** **alert 770,"…","…., [sellerh] > [buyerh], initiator = [initiatorh]"** **end if** **end on**
buyert *(may not be available)*	The buying trader during an on trade rule (see section 7.4 "Current Object Functions")..
sellert *(may not be available)*	The selling trader during an on trade rule (see section 7.4 "Current Object Functions")..
initiatort *(may not be available)*	The trader who initiated the current trade. Only defined inside an **on …** when clauses relating to trades (eg: **on trade**, **on mkttrade**).
buyerc *(may not be available)*	The buying client during an on trade rule (see section 7.4 "Current Object Functions")..
sellerc *(may not be available)*	The selling client during an on trade rule (see section 7.4 "Current Object Functions")..
initiatorc *(may not be available)*	The client who initiated the current trade. Only defined inside an **on …** when clauses relating to trades (eg: **on trade**, **on mkttrade**).
primaryhouse *(may not be available)*	This returns true or false, depending on whether the current house is a primary house or not.
major_turnerh **major_turnerh(time)**	The house who did the most trading (buying + selling) in terms of trade value, in the current security on the current date. For example: To identify the house who traded the most in the top 10 securities at the end of the day: **at dayend** **per security** **if rank(alertdate – 1 days) <= 10 then** **print "[major_turnerh] did the most trading in [security] today."** **end if** **end per** **end at** If you specify a time parameter, then it will consider trading from just that time onwards until the current time. You can also specify a negative time parameter, which will be interpreted in the usual way i.e. as an offset against the current time, e.g. –0:05 means the last 5 minutes of trading. Also, to use this function and others of the major_* function family below, you must be inside a particular day. So using these functions in an 'at end' or 'at start' when clause will give you undefined values, this is because the last day evaluated has been closed in an 'at end' or has not been opened in an 'at start' when clause. Use them in an 'at dayend', 'at daystart', 'on trade', etc when clauses instead.

Function	Description
major_buyerh **major_buyerh(time)**	The house who net bought the most (buying - selling), in terms of trade value, of the current security on the current date.
major_sellerh **major_sellerh(time)**	The house who net sold the most (buying - selling), in terms of trade value, of the current security on the current date.
major_turnert **major_turnert(time)**	The trader who did the most trading of the current security.
major_buyert **major_turnert(time)**	The trader who net bought the most of the current security.
major_sellert **major_sellert(time)**	The trader who net sold the most of the current security.
major_turnerc **major_turnerc(time)** *(may not be available)*	The client who did the most trading of the current security.
major_buyerc **major_buyerc(time)** *(may not be available)*	The client who net bought the most of the current security.
major_sellerc **major_sellerc(time)** *(may not be available)*	The client who net sold the most of the current security.
mplid(house) **mplid(trader)** **mplid(client)**	Obtains the mplid. This helps in the case where participant is renamed. The mplid remains the same.
major_turnerx **major_turnerx(time)** *(may not be available)*	The client who did the most trading of the current security. Note: this is for markets where tagfields represent clients.
major_buyerx **major_buyerx(time)** *(may not be available)*	The client who net bought the most of the current security. Note: this is for markets where tagfields represent clients.
major_sellerx **major_sellerx(time)** *(may not be available)*	The client who net sold the most of the current security. Note: this is for markets where tagfields represent clients.

7.8 Order-book functions

Function	Description
Askordercountatstep(number)	The current total ask order count at price step N
askpriceatstep(number)	Returns the price of the ask orders at step N where best ask = 0.
bidpriceatstep(number)	Returns the price of the bid orders at step N where best bid = 0.
PriorityAsk	The order id of the next ask to trade.
PriorityBid	The order id of the next bid to trade.

bidvolbetween(price1,price2)	The volume of bids in the current security having a price in-between price1 and price2 (inclusive). This function was formerly known as 'bidbetween'.
	NOTE: If **price1** is $0.00, then this means every bid order below **price2**. If **price2** is $9999.99, then this will mean every bid order above **price1**.
	For example:
	To print the volume of bid orders between the current midpoint price and 5% below the midpoint price, type:
	`at 12:00`
	`per security`
	`print "Volume of bids between [mpprice] (midpoint) and [mpprice - (0.05 * mpprice)] = [bidbetween(mpprice, mpprice - (0.05 * mpprice))]"`
	`end per`
	`end at`

askvolbetween(price1,price2)	The volume of asks in the current security having a price in-between price1 and price2 (inclusive). This function was formerly known as 'askbetween'. **NOTE**: If **price1** is $0.00, then this means every ask order below **price2**. If **price2** is $9999.99, then this will mean every ask order above **price1**.
bidvalbetween(price1,price2)	The value of bids in the current security having a price in-between price1 and price2 (inclusive). **NOTE**: If **price1** is $0.00, then this means every bid order below **price2**. If **price2** is $9999.99, then this will mean every bid order above **price1**.
askvalbetween(price1,price2)	The value of asks in the current security having a price in-between price1 and price2 (inclusive). **NOTE**: If **price1** is $0.00, then this means every ask order below **price2**. If **price2** is $9999.99, then this will mean every ask order above **price1**.
houseask(house,volume)	The first price such that this house has at least 'volume' units on order to sell at or below the price.
houseask(house, volume, flagsfilterstring, fieldsfilterstring)	The first price such that this house has at least 'volume' units on order to sell (which satisfy the specified filters) at or below the price. Examples of flagsfilterstring: ➢ "-OF-od" (exclude off market and oddlot orders) ➢ "+IA" (include inactive orders only) Examples of fieldsfilterstring: ➢ "C=ClientA\tClientB" (include orders whose tagfield C has values ClientA **or** ClientB) ➢ "C=ClientA\tClientB\nDT=1:Proprietary" (include orders whose tagfield C has values ClientA **or** ClientB, **and** whose littlefield DT has value 1:Proprietary) \t and \n in this context are called delimiters. They are characters that lets the alice engine know to "stop here" when obtaining tokens and string values. You may also use \1 in place of \t and \2 in place of \n.
housebid(house,volume)	The first price such that this house has at least 'volume' units on order to buy at or above the price. For example: To print each house's best bid and best ask in the top 1 security at the

Function	Description
	end of trading, type: **at 16:00** **per security** **if rank(alertdate – 1 days) = 1 then** **per house** **print "best bid = [housebid(house,x1)],** **best ask = [houseask(house,x1)]"** **end per** **end if** **end per** **end at**
houseask(house, volume, flagsfilterstring, fieldsfilterstring)	The first price such that this house has at least 'volume' units on order to buy (which satisfy the specified filters) at or above the price. Examples of flagsfilterstring: ➢ "-OF-od" (exclude off market and oddlot orders) ➢ "+IA" (include inactive orders only) Examples of fieldsfilterstring: ➢ "C=ClientA\tClientB" (include orders whose tagfield C has values ClientA **or** ClientB) ➢ "C=ClientA\tClientB\nDT=1:Proprietary" (include orders whose tagfield C has values ClientA **or** ClientB, **and** whose littlefield DT has value 1:Proprietary) \t and \n in this context are called delimiters. They are characters that lets the alice engine know to "stop here" when obtaining tokens and string values. You may also use \1 in place of \t and \2 in place of \n.
housespread(house,volume)	Equal to **houseask(house, volume) - housebid(house, volume)**.
housespread(house, volume, flagsfilterstring, fieldsfilterstring)	Equals to **houseask(house, volume, flagsfilterstring, fieldsfilterstring) – housebid(house, volume, flagsfilterstring, fieldsfilterstring)**
house_bidbetween(**house, price1, price2)**	The volume of bid orders the current or specified house has in the current security, between the two prices specified (inclusive). **NOTE**: If **price1** is $0.00, then this means every bid order below **price2**. If **price2** is $9999.99, then this will mean every bid order above **price1**. For example: To determine the percentage of each house's bid volume up to 5% below the current midpoint price, type: **[house_bidbetween(house,mpprice,mpprice - (0.5 * mpprice)] %** **[house_bidbetween(house,$0.00,mpprice)]**
house_askbetween(**house, price1, price2)**	The number of units this house has in ask orders in the current security, between the two prices given (inclusive). **NOTE**: If **price1** is $0.00, then this means every ask order below **price2**. If **price2** is $9999.99, then this will mean every ask order above **price1**.
traderask(trader, volumel) **traderbid(trader, volume)** **traderspread(trader,volume)** **trader_bidbetween(trader, p1, p2)** **trader_askbetween(trader, p1, p2)** *(may not be available)*	Trader versions of the above functions.
AskValUpto(limitprice, limitvolume)	This calculates how much it would cost to take out the entire ask side of the orderbook up to the 'limitprice' and up to the 'limit volume'. The value divided by **AskVolupto** returns the **vwap** of putting in a large immediate order.
BidValDownto(limitprice, limitvolume)	This calculates how much it would cost to take out the entire bid side of the orderbook up to the 'limitprice' and up to the 'limit volume'. The value divided by **BidVoldownto** returns the **vwap** of putting in a large immediate

Function	Description
	order.
AskVolUpto(limitprice, limitvolume)	This calculates how much volume is required to take out the entire ask side of the orderbook up to the 'limitprice' and up to the 'limitvolume'.
BidVolDownto(limitprice, limitvolume)	This calculates how much volume is required to take out the entire bid side of the orderbook up to the 'limitprice' and up to the 'limitvolume'.
liqdown **liqdown(time)**	(a) The market depth of bids at the current time, or (a) The market depth of bids at a given time. In both cases it corresponds to the **SMARTS** 'liqdown' metric.
liqup **liqup(time)**	(a) The market depth of asks at the current time, or (b) The market depth of asks at a given time. In both cases it corresponds to the **SMARTS** 'liqup' metric.
entrytime	Inside a 'per order' or "on amend command to query the time an order was entered
Bidordercountatstep(number	The current total bid order count at price step N where 0= best bid.
findorderprice(number)	Gives you the price of an order identified by orderid.

7.9 Field-extraction functions

Function	Description
askorderfield(string)	Look up this field in the current sell order.
bidorderfield(string)	Look up this field in the current buy order.
namefieldvalue("market", "entity type", "code", "tag", date)	Look up this entity's field value from the specified market, date and entity type. Entity type can be one of: security, house, trader, client, index, sechouse, sectrader, secindex, settlepot, issuer, marketinfo, houseclient.
namefieldfirstdate(entity type, code)	Returns the date the entity first appeared in the reference database.
namefieldlastdate(entity type, code)	Returns the date the entity was delisted or the date when the entity last changed a reference field.
namefieldnextchangedate(entity type, code, date) **namefieldnextchangedate(entity type, code, field, date)**	Returns the date of the next reference field change for the specified entity or the date the specified reference field was changed. This function would return the same date as the as the specified date if there was a reference field change on the date specified as it is an inclusive operation.
namefieldprevchangedate(entity type, code, date) **namefieldprevchangedate(entity type, code, field, date)**	Returns the date of the last reference field change for the specified entity, or the date the specified reference field was changed. This function would return the same date as the as the specified date if there was a reference field change on the date specified as it is an inclusive operation.
orderfield(string)	Look up this field in the current order. Works for both tagged fields and little fields.
fieldname(string)	Returns the configured long name of the specified tagfield or littlefield code
fieldtrans(string)	Returns the transaction types that the specified tagfield is configured to be used for
fieldfortrans(string, "trans type")	Returns true if the specified tagfield is configured to be used for the specified transaction type ("O", "C", "D", "T").
fieldcode(number)	Return the configured littlefield code for the n'th littlefield
fieldvaluemax(string)	Return then number of configured values for the specified littlefield
fieldtype(string)	Returns the data type of the specified tagfield
fieldnum	Returns the number of littlefields available for the current market

securityfield("tag")
securityfield(security," tag")
	Looks up the security-field for the field tag specified and returns the field value as a string.
	NOTE: A complete list of field tags for securities are listed in Appendix A of the SMARTS User Manual.
	For example:
	Assume that each security has been assigned an industry tag (ID) ranging from 1 to 5 depending on which industry the security is in.
	a) To evaluate an alert rule only for securities falling into the industry 1 category, type:
	`on trade`
	`  if securityfield("ID") = "1" then`
	`  if ....`
	`  alert ...`
	`  end if`
	`  end if`
	`end on`
	b) To print the value of a security field tag, type:
	`on trade`
	`  if ...`
	`  alert 790,"...","... [security] which is in the industry [securityfield("ID")]".`
	`  end if`

Function	Description
	end on **NOTE**: It is only necessary to specify the security inside the **securityfield(security,"<tag>")** function if you want to refer to a tag of a security other than the default security or security currently being evaluated.

Function	Description
securityvfield("tag") **securityvfield(security, "tag")**	Similar to the above except it returns a number. (If the field is not a numeric field, then the result is undefined.) To use this field as part of calculations, you might need to convert it to a price, value, volume or percent: 1. multiply the ordinary number by $1 to convert it into a value x1 to convert it into a volume $1.00 to convert it into a price 100% to convert it into a percent. For example: Assume that each security has an issued capital (IS) tag which simply represents the security's issued capital. To create an alert rule which evaluates the percentage of issued capital traded, type: on trade if volumeall(0:00) % (**x1 * securityvfield("IS")**) >= 2% then alert … end if end on
housefield("tag") **housefield(house,"tag")**	Looks up the house-field for the field tag specified and returns the field value as a string. Usage of this function is similar to **securityfield(…)**.
traderfield("tag") **traderfield(trader,"tag")** *(may not be available)*	Looks up the trader-field for the field tag specified and returns the field value as a string. Usage of this function is similar to **securityfield(…)**.
clientfield("tag") **clientfield(client,"tag")** *(may not be available)*	Looks up the client-field for the field tag specified and returns the field value as a string. Usage of this function is similar to **securityfield(…)**.
settlepotfield("tag") **settlepotfield(house,"tag")** *(may not be available)*	Looks up the settlepot-field for the field tag specified and returns the field value as a string. Usage of this function is similar to **securityfield(…)**.
namefieldtagname("entity type", "tag")	Looks up the tag long name based on the <entity>_nametag market configuration.
namefieldtagvalname("entity type", "tag", "tag value")	Looks up the long name of the specified "tag value" of the specified "tag" based on the <entity>_nametag market configuration.
secindexfield(security,index,string)	Look up field T of this security-in-index
mplid("entity type", "code") **mplid("entity type", "code", date)** **mplid("market", "entity type", "code", date)**	Converts a code of an entity to its mplid. If date is not specified, the current date is used. If market is not specified, the current market is used.
	Converting strings to entities
security(string) **security(string, date)**	Converts a string representing a security code into a security. Use the code current date if date is not specified.
house(string) **house(string, date)**	Converts a string representing a house code into a house. Use the code current date if date is not specified.
trader(string) **trader(string, date)**	Converts a string representing a trader code into a trader. Use the code current date if date is not specified.
client(string) **client(string, date)**	Converts a string representing a client code into a client. Use the code current date if date is not specified.
index(string)	Converts a string representing a index code into a index. Use the code current date if date is not specified.

Function	Description
index(string, date)	
	Converting mplids to entities
security(mplid)	Converts an mplid into a security
house(mplid)	Converts an mplid into a house
trader(mplid)	Converts an mplid into a trader
client(mplid)	Converts an mplid into a client
index(mplid)	Converts an mplid into an index

7.10 Position and Profit functions

The following functions require you to have selected a particular house, trader or client, plus a particular security. These objects can be selected by either a **when clause**, the **per** statement, or the **with** statement. These functions then refer to the position or profit of that participant in the current security.
Note that the **on trade** clause does *not* set the current house or trader or client, because it is ambiguous to use the buying participant or selling participant. Instead, the programmer should break the rule into two halves, and use the **with** statement with the **buyerh, buyert** or **buyerc** functions, and then with the **sellerh, sellert** or **sellerc** functions (as described below).

Function	Description
pos_shares_in	Number of shares the current participant bought in the current security on the current day.
pos_shares_out	Number of shares the current participant sold in the current security on the current day.
pos_net_volume	Equal to **pos_shares_in - pos_shares_out**.
pos_cash_in	Amount of money the current participant got by selling units of the current security on the current day.
pos_cash_out	Amount of money the current participant paid when buying units of the current security on the current day.
pos_net_value	Equal to **pos_cash_out - pos_cash_in**.
pos_buy_tcount	Number of buy trades the current participant made in the current security on the current day.
pos_sell_tcount	Number of sell trades the current participant made in the current security on the current day.
pos_net_tcount	Equal to **pos_buy_tcount - pos_sell_tcount**.
pos_profit	Current participant's unrealised profit in the current security on the current day. (It values the participant's change in portfolio based on the last traded price).

7.11 Futures and Options functions (only applicable to derivatives markets)

Function	Description
issuedate	The date of issue of the current security. This comes from the "ID" tag in the security.name file, or if not present, it takes the first date this security is mentioned in the security.name file.
maturitydate	The maturity date of the current security. Undefined for equities and instruments that don't mature.
putorcall	Returns "P" or "C"
strikeprice	Returns the strike price of this option
underlying	The underlying security
underlyingprice	The trueprice of the underlying security
unitfacevalue	The number of underlying units represented by one unit of the derivative
syield(price) *(for bonds)*	The simple yield of this instrument for this price. The yield is annualised. It returns a percentage figure.
eyield(price) *(for bonds)*	The effective yield of this instrument for this price. The yield is annualised. It returns a percentage figure.
underlying	The equity or instrument underlying the current security (option/future).
underlyingprice **underlyingprice(time)** **underlyingprice(date)**	The trueprice (usually equals last-traded-price) of the underlying instrument as at (a) the current time, (b) the time specified, (c) the close of trading on the date specified.
measurevolatility (security, number)	Calculates the volatility of the specified security over the specified number of days. **NOTE**: (a) you generally put the underlying security, not the current security, as the 1st parameter, eg.: **measurevolatility(underlying, 180 days)**; (b) the date-range goes backwards from the current date by the number of days specified (calendar days, e.g. 92=3 months); (c) the result is an annualised percentage figure. The formula is: Standard deviation of returns, considering only trading days, * sqrt(248). The 248 is because we assume there to be 248 trading days per year: 260 non-weekends and 12 public holidays. Even if this is not exactly accurate, it cancels out when we use volatility in the binomial formula because we use the same figure of 248 there. (d) The 'standard deviation' is actually a weighted standard deviation: returns over the past 30 calendar days carry twice the weight of returns before that. (e) Returns are official closeprice to official closeprice. We use log-returns i.e. log(P2/P1) which is almost identical to (P2-P1)/P1 for small returns. (f) We skip weekends and non-trading-days and days where the official closeprice is undefined for any reason (e.g. the stock is not yet listed). The remaining set of defined close-prices then form an array, and our array of returns is taken from one element of this array to the next. (g) We exclude any outlier return. An outlier return is one which is more than Z standard deviations from the mean, where Z = sqrt(N) * 0.5. This rarely leads to exclusions but will save you if there is a real outlier.
binomial(price)	The Binomial price of an options contract, calculated on the basis of this underlying price. The other inputs to the Binomial pricing model are inserted automatically. In particular, the volatility is calculated using the above function on the underlying security over 90 days. The answer you get will be equivalent to the answer the Black&Scholes model gives provided it's a case where the Black&Scholes model applies, namely a European option with no dividends.
binomial(price,number)	The Binomial price of an options contract, computed on this underlying price and this volatility number. Note: this function was formerly named blackschole.

Function	Description
	For example: **binomial(underlyingprice, measurevolatility(underlying, 180 days))**
theoreticalprice(price, percent)	Uses binomial in reverse to return the underlying price we expect with an option of price p and volatility V..
maturitydate	The date of maturity of the instrument.
strikeprice	The strike price of an options instrument.
putorcall	Returns "P" if the current security is a put (option to sell) and "C" if the current security is a call (option to buy). It is undefined if the current security is not an option instrument. For example: **if putorcall = "P" then … end if** (Make sure you use a capital "P" to compare it with).
impliedv(price)	The implied volatility of this price. This is the inverse function to the **binomial(price)** function.
option_delta	BlackScholes/Binomial delta : sensitivity to price change
option_gamma	BlackScholes/Binomial gamma: sensitivity of delta to price change
option_theta	BlackScholes/Binomial theta: sensitivity to time decay
option_vega	BlackScholes/Binomial vega: sensitivity to changes in volatility

Note that the accuracy of these pricing models depend on many things. For example, you must have the full specification of each options contract, you must have sufficient data on the underlying instrument to measure meaningful past volatility figures, you should adjust for expected dividends, and you should have a feed of interest rate data.

7.12 CAPM functions

Function	Description
capm_beta(security,index) **capm_alpha(security,index)**	The CAPM model constructs "expected stock prices" from an index time-series. The model says that if S is the stock price and X is the index price, then: S = beta*X + alpha …where 'beta' and 'alpha' are coefficients derived from past price data. If the CAPM model is used to produce better price alerts, then the 'beta' is more important than the 'alpha'. A price change of 1% in the index should result in a price change of 'beta'% in the security.

7.13 Alerts query functions

Function	Description
AlertCountAll(date) **AlertCountAll(date, date)**	The total amount of alerts for the given date or between date1 to date2 (inclusive) for all securities.
AlertCount(date) **AlertCount(date, date)**	The total amount of alerts for the given date or between date1 and date2 (inclusive) for the current security.
GetAlertStatistics(string, date)	Initiates the alert accounting for the market and date given.
AlertField(string)	To be used in conjunction with GetAlertStatistics and GetNextAlert functions. Returns named alert field as a string or undefined. The 'field' to be used must be one of the field names in the table below.
GetNextAlert	To be used in conjunction with GetAlertStatistics function. Gets the next alert ready for the 'AlertField' call. This function returns 0 if there are no more alerts in the queue.
AlertComment	To be used in conjunction with GetAlertStatistics and GetNextAlert functions. Return current alert comment as a string.
GetNextComment	To be used in conjunction with GetAlertStatistics and GetNextAlert functions. Gets the next alert comment from the current alert ready for the 'AlertComment' call. The function returns false if there are no more alert comments in the queue.

The following table contains a definitive list of field names available to be used with the 'AlertField' function.

Field	Description
id	ID of alert
startdate	Start date of Alert
date	Date of alert
starttime	Start time of alert
time	Time of alert
code	Alert code
security	Security of alert
intensity	Intensity of alert
market	Market for current alert
extrafolder	Extra folder of alert
folder	Folder of the alert
house	Alert house
trader	Alert trader
client	Alert client
viewer	Default viewer for alert
commandline	Command line for alert
shorttext	Shortext for alert
user	Current user for alert
fromuser	Previous user for alert
lastuser	The previous user who analysed the alert
lastcomment	The very last comment noted in the alert
longtext	Long text for alert

Field	Description
refcount	Referral count for alert (Customer specific)
ref_status	Referral status of alert (Customer specific)
ref_open_user	User that sent referral for alert (Customer specific)
ref_open_date	Date the referral was sent (Customer specific)
ref_return_user	The last user to respond to the referral (Customer specific)
ref_return_date	Date the response was sent for the referral (Customer specific)
ref_open_time	Time the referral was sent (Customer specific)
ref_return_time	Time the response was sent for the referral (Customer specific)
ref_number	The unique referral number (Customer specific)

Please note, that configurable fields may also exist and maybe queried using the AlertField function as well, however users are required to know the exact name of the configurable field.

The following is an example of Alerts query functions usage.

```
at start
        print "Alerts statistics for  31/07/2007"
        print "GetAlertCountAll [AlertCountAll(31/7/2007)]"
        print "GetAlertStatistics [GetAlertStatistics(31/7/2007)]"
        print ""

        for ; GetNextAlert(); do
                print "[AlertField("id")]"
                print "[AlertField("id")]"

                print "ALERT COMMENT TRAIL"
                print "[AlertField("lastcomment")]"
                print ""

                print "ALERT COMMENTS"
                for ; GetNextComment(); do
                        print "AlertComment [AlertComment()]"
                end for
                print "END\n"
        end for
        exit
end at
```

8. EXPRESSIONS

8.1 Introduction

An expression essentially means a 'formula'. Formulae are built up by combining constant values of the various *Types*, *Functions* and *Operators*. The **(** and **)** symbols can be used to bracket expressions in the normal mathematical sense. Normal precedence rules also apply to expressions, eg.:

3 + 4*5 means 3 + **(**4*5**)**

A list of the different *Types* and *Functions* are provided in sections 6 and 7 respectively. A list of the different type of *Operators* is provided below.

8.2 Operators

Operator	Description
+	addition
-	subtraction
*****	multiplication
/	division
=	equality test
<	less than test
>	greater than test
<=	less than or equal test
>=	greater than or equal test
!=	not-equals-to
and	logical and
or	logical or
not	logical not
<-	add value to the distribution (mainly for the purpose of generating a histogram to determine a benchmark)
+=	add value to the total
-=	deduct value from the total

NOTES:
The *string* > *string* function allows for comparison operators in addition to the equals/not equals operators with strings. All string comparisons are done in a case-sensitive way e.g. "ABC" < "abc" is true.
You can merge two distributions with the "let X += Y" statement. This will put all the observations together into one big distribution.
You can't add two strings: if you want to concatenate strings, then use the feature of embedded expressions, e.g.: let X = "[Y][Z]"

8.3 Undefined Values

A useful feature of *Alice* is its support for undefined values. Any expression may give an undefined value. For example, certain functions may return undefined values if the necessary information cannot be obtained. If an undefined value is combined with other values using the operators, the result is another undefined value. Undefined values are printed as **(undefd)**.

For example, suppose we are currently looking at a security for which average daily trading benchmark information is unavailable (which may happen in the case of a newly floated security), the following expression will return:

print avdailyval	returns	**(undefd)**
print avdailyval * 3 + $10	returns	**(undefd)**
print $300 < avdailyval	returns	**(undefd)**
print $300 < avdailyval or 3 < 4	returns	**true**
print $300 < avdailyval or 3 > 4	returns	**(undefd)**

Where an **if** statement has as its condition an undefined expression, *neither* the **then** statement *nor* the **else** statement will be executed. For example, the following will print nothing if **avdailyval** is undefined:

```
if avdailyval < $1000 then
        print "A small security"
else
        print "A big security"
end if
```

This is appropriate here because it is not known if the security should be classified as big or small. The special function **defined** can be used to determine if a given expression is defined or not. The **defined** expression itself is always guaranteed to be defined. For example:

```
if defined(avdailyval) then
        print "We have benchmark information"
else
        print "Benchmark avdailyval is missing."
end if
> Benchmark avdailyval is missing.
```

Alternatively, an **undefined** expression or **not defined** expression may be used. For example:

```
if undefined(avdailyval) then
        print "We have NO benchmark information"
end if
```

The **and** & **or** operators work in an intuitive way with undefined values. Eg.:

<undefined> **and** false = false,

whereas

<undefined> **and** true = <undefined>.

9. FORMATTING

9.1 Spacing

An *Alice* program can have as many blank lines as desired. And wherever a space is required (for example to separate two keywords) there is no limit to how many spaces or <TAB> characters may be used. This is useful for giving the program an attractive layout.

NOTE: Inserting <TAB> between words in the *"fulltext"* section of the **alert** *statement* may result in large spaces appearing in between the words. If the *"fulltext"* string is too long, hit <ENTER> to continue on the next line instead of <TAB>.

As in the C language and most other programming language, certain conventions apply when using the *Alice* language. Statements should be indented according to which other statements they are embedded inside. For example:

```
at start
        with ^BHP
                print security
                if avdailyval > $1000 then
                        print "BHP has big trades"
                        alert 101, "BigADV", "Big avdaily val"
                end if
        end with
end at
```

An unusual feature of the *Alice* language is that strings can be broken up into multiple lines. Where this occurs, the newline characters (ie. the invisible characters that indicates when one line finishes and the next one begins), and the following indentation, will be removed. This is useful for **alert** *statements*, which usually have long and complex strings within them.
For example:

```
alert 101, "HIGH MT VOLUME", "HIGH MEDIUM TERM VOLUME: Volume over
        [mt_period] hour [volume(-mt_period)] compared to benchmark of
        [mtvol[security]]."
```

In this example, newline characters at the end of the first and second line after the words **over** and **of** respectively, will be removed.
Note however, that unlike **alert** *statements*, the syntax following the definition of user parameters cannot be broken up into multiple lines.

9.2 Comments

At any point in an *Alice* program, the programmer can insert comments. A comment is a note to the reader explaining the nearby code. Comments are ignored as the program executes, but are useful for reminding the programmer of various matters.
There are three ways of inserting comments in an *Alice* program:

- text contained within the symbols **/*** and ***/** (like C comments)
- text following the symbol **//**, up to the end of the line (like C++ comments)
- text following the symbol **#**, up to the end of the line (like UNIX shell comments).

NOTE: The exception to the # symbol denoting a comment is when it is used as a macro. Refer to section 11.4 for more about Macros.

Below is an example of how comments are inserted in an *Alert Rule*.

```
/*
----------Super Ramping Alert----------
This alert considers whether a security have been ramped and consider the change in price
between the ramped closing price and the opening price the next morning.  If the price fell the
following morning by the ramped percentage specified in the user parameters then an alert is
fired.
*/

# Alert creation date: 27/8/97
# Alert amendment date: 27/8/97


on trade
    if dayofweek = 0 then
        let range = 3 days
    else
        let range = 1 days
    end if
// 0 represents Mondays, 1 represents Tuesdays, etc.  If the trade occurred on the Monday, we
// will want to determine the change in price since Friday thus we let the variable range = 3
days.
    if ramped[security] then
        if change(price, lastclose[security]) < - ramping then
            alert 510, "SUPER RAMPING", "SUPER RAMPING: price yesterday increased
                [change(lastclose[security], unramped[security])] from [unramped[security]] to
                [lastclose[security]] during the last 10 minutes and fell [change(price,
                lastclose[security])] from [lastclose[security]] to [price] this morning",
                    date0 = date - range,
                    commandline = "-time {9:45 16:15}",
                    reissue = "510SD"
        end if
    end if
end on
```

10. WRITING GOOD ALICE PROGRAMS

10.1 Avoid repeating slabs of code

When constructing an alert rule, often it is necessary to have various variations on the alert rule. For example, liquid securities might have slightly different benchmarks to illiquid securities. Or sometimes the spread is very large, and sometimes it might be undefined, but both situations are similar.
Beginners to the *Alice* language often deal with this need for variations by writing large numbers of very similar alerts, each differing in just a small element.
This can be avoided by making liberal use of temporary variables. A temporary variable is just an ordinary variable which is used to temporarily store data, not to accumulate data over the lifetime of the run. For example, rather than:

```
on trade
    if liquid[security] then
        if volume > largetradeval[security] then
            alert 101, ...
        end if
    else
        if volume > illiq_LTV then
            alert 102, ...
        end if
    end if
end on
```

It would be better to write:

```
on trade
    if liquid[security] then
        declare let benchmark = largetradeval[security]
    else
        declare let benchmark = illiq_LTV
    end if
    if volume > benchmark then
        alert 101, ...
    end if
end on
```

You can even have a different alert-code without duplicating the **alert** *statement*, for example:

```
on trade
    if liquid[security] then
        declare let benchmark = largetradeval[security]
        declare let alcode = 101
    else
        declare let benchmark = illiq_LTV
        declare let alcode = 102
    end if
    if volume > benchmark then
        alert alcode, ...
    end if
end on
```

Sometimes you need to check some condition every time an order is either entered or amended, or deleted/traded, or at 3 specific times during the day. This can be done with a single "when-clause", i.e. without needing to duplicate any code. For example:

```
on trade, delord
    /* check market-makers' spreads */
end on

at 10:15, 15:45
    /* check something else */
end on
```

10.2 Avoid slow programs

While it is not possible to write *Alice* programs that loop forever[1], it is possible to write extremely slow programs. To avoid this, you should:

- Never put a **per** statement inside an **on** clause. This will be very slow, and is almost invariably not what the programmer intended.
- It is preferable to use **per security, house** rather than **per security per house**. This will give you all active (security, house) pairs, meaning that the house has made at least one trade in that security. However, you cannot of course do this at the beginning of the day.
- If you have **per** statements inside an **every** clause, you should try not to have small intervals in the **every** clause, especially if you are using **per security, house**.

For example:

```
on trade                        /* This is almost certainly a mistake */
    per security
        ...
    end per
end on

every 1 mins                    /* This will be very slow. */
    per security, house
        ...
    end per
end every
```

[1] Except with the 'for' loop in certain cases

10.3 Good alert design

This section gives you guidelines at a higher level: how to design good alerts.

10.3.1 Be aware of sample sizes

If you are calculating averages or histogram cutoff levels, then you must be aware of your sample size. For example, if you define "largetradeval" to be the 95% cutoff level in the histogram of trade sizes, and you are indexing the set of trades by security, and the security is very illiquid, then you will get very unstable and useless benchmark numbers. To avoid this situation, you should consider one of the following:

a) group all securities together, i.e. don't index by security;

b) group securities together to get bigger samples;

c) have different alerts for illiquid securities; or

d) reduce the cutoff e.g. to 80%, and instead have a multiplier to make it larger.

10.3.2 Don't include the alerting period in the benchmark

If you are doing long term alerts that span several days, it is important not to include the alerting period in the benchmark. For example, suppose you are creating a 7-day volume alert. If your benchmark goes up to the day before the alerting period, then you will include 6 of the 7 alerting days in the benchmark. This will make the benchmark go up with the metric itself, and give you very few alerts, or give you completely spurious alerts.

10.3.3 Price alerts: use the correct price measure

There are 3 ways to measure the current 'equilibrium price' of a security: (a) the last trade price, (b) the bid/ask midpoint price and (c) the **SMARTS** 'true price' metric. You should understand the disadvantages and advantages of each.

The 'last trade price' suffers the disadvantage that it can sometimes be overtaken by bids or asks without any trade occurring, so that e.g. the last trade price is less than the current bid price. In this case, the last trade price would not be an accurate indication of the current 'equilibrium price' of the security. Also, the last-trade-price as measured by the *Alice* function 'lastprice' does not carry forward the trade price from previous days, and hence can be undefined even when the midpoint price is defined. Also, this measure can give a false indication of volatility: If there is a constant best-bid and best-ask price, but the trade price oscillates between the two sides, then the 'last trade price' will falsely indicate a volatile market.

The 'bid/ask midpoint price' is good for solving the last problem, but it suffers the disadvantage that in thinly traded (illiquid) securities, it can be determined by orders which are too far from the market to be indicative of the actual price. Also, while it is sometimes defined when last-trade-price is not defined, the reverse can also be true. Another disadvantage is that at moments when the market is overlapping, e.g. in the instance between a market order and the corresponding trade, it is not as well defined. This is an issue that arises when writing alerts. An attempt is made to predict the mid-point price after the order has gone through, so that you can use 'mpprice' inside an "on entord" clause, but if this prediction is false than it can lead to an unusual price.

The **SMARTS** 'True Price' metric (see the 'trueprice' function) was designed to improve on the other two price measures. It is defined as the last-trade-price, unless the last-trade-price has been overtaken by a bid or ask - in which case it is set to the price of the overtaking order. Also, at the beginning of the day before the first trade, it is set to the midpoint price. Note that it ignores off-market trades. In very thinly traded securities, where there is a bid but no ask or vice versa, and no last-traded-price, it is set to the price of the order on the one side of the market. In practice, the 'True Price' is usually equal to the last-trade-price.

10.3.4 Use a combination of absolute and relative comparisons

When creating price alerts, you must have a combination of relative versus absolute conditions. For example, if an 8 cent stock rises to 9 cents, this is a 12% increase in price, but really should not be considered unusual. On the other hand, a 12% increase in a liquid security with a high price relative to the tick size, would be considered very unusual over a short term. On the other hand, a 5 cent increase in the 8 cent stock would be unusual, but a 5 cent increase in a stock trading around $2.00 would not be unusual.
So you should design an alert as follows:

```
if change(price1, price2) > PARAM1
    and price1 > price2 + PARAM2 then ...
```

Often it is good to have two alerts with different relative absolute and relative parameters. For example:

```
if change(price1, price2) > 10% and
    price1 > price2 + $0.02 then
        alert 101, "PRICE RISE (LOW PRICE)", "..."
else
if change(price1, price2) > 5% and
    price1 > price2 + $0.10 then
        alert 102, "PRICE RISE (HIGH PRICE)", "..."
```

10.3.5 Consider time-of-day effects

You sometimes need to consider time-of-day effects. For example, often there is a lot of trading at the beginning and the end of the trading session. This means that short-term volume alerts around these periods should be eliminated altogether, or at least given higher thresholds than short-term volume alerts during the remainder of the trading day. The **SMARTS** program **ACCESS** can help you identify these transitions between different types of trading.

10.3.6 Deal with illiquid securities appropriately

Illiquid securities often need different alerts to liquid securities. For example, if a stock trades only a few times a week, then most price alerts or volume alerts will be inappropriate. You should instead either group illiquid securities into groups, or use alerts with absolute thresholds (e.g. you type in a number: x25000) rather than statistical thresholds calculated on the benchmark period.

11. ADVANCED ALICE FEATURES

11.1 User Defined Functions

User defined functions are a new feature to Alice and are useful in the sense that now advanced users have more control if certain functions aren't available in the current SMARTS ALICE function library. Also, user defined functions may help in making Alice programs more tidy and compact.

11.1.1 Different Types of Functions

Basically, there are 2 types of functions, outlined below (keywords are in **Arial Bold**):

1. A function that returns something

- General function declaration syntax

```
function function_name(param1 : paramater1_type, param2 : paramater2_type) : return_type

    [ ALICE statements & declarations ]

    return something_to_return

end function_name
```

Function identifiers.

Whatever is returned via this return statement must be of the same type as in the function declaration.

- Examples of function declaration:

```
function Foo() : value

    with ^CBTP
        return value(date -1 days)
    end with

end Foo
```

The type of the something to be returned by the functions

'**return**' keyword indicating this is where this function ceases execution and returns something to the function caller.

```
function totals(v : volume, p : price ) : value

    declare let tot = v * p

    if v > x0 then
        return  tot
    else
        return $0
    end if

    return $0

end totals
```

Parameter list. When calling this function you are expected the pass it input as specified in this list

Any variables declared inside functions are local to the function. Once the function ceases execution this variable can no longer be accessed. Refer to the 'Scope' issue in this section for more on local variables.

Even though all '**if**' paths have return statements, you must have a default return at the end of the function. This is to guard against cases where the condition inside the '**if**' statement is undefined, then it falls through both paths of the '**if**' statements. (ie. Skips the if-else statement all together if the condition v > x0 is undefined)

- Examples of calling the functions :

```
at start
    declare tot : price
    declare pp : price
    declare val : value
    declare vol : volume

    Foo
    Foo()
```

When a function has no parameter list, it may be called either way, with () or without brackets

```
    val = Foo
    print "\nval : [val]"
```

You may catch whatever the function returns, or simply ignore it like the above 2 function calls

```
    totals(x40000, $12.00)
```

Passing input, note that the types of input must be the same as the ones in the parameter list of the function declaration

```
    print "\nThe total price is : [totals(x41000, $1.23)]"
```

You may simply print whatever the function will return in the usual manner

```
    vol = x55000
    pp = $3.98
    tot = totals(vol, pp)
```

You can pass in other variables or complex expressions as input to a function, not just constants

```
    print "\nBuying [vol] units at [pp] per unit, total is [tot]\n"
end at
```

2. A function that doesn't return anything

- General function declaration syntax

Function identifiers.

```
function function_name(parameter1 : paramater1_type, parameter2 : paramater2_type)

    [ ALICE statements & declarations]

end function_name
```

- Examples of function declaration

```
function Hello(name : string)
    print "Hello there [name], how are you today?"
end Hello
```

All input parameters are also local to the function. Any changes you attempt to do to them will be lost once the function ceases execution.

```
function printTotal(s : security, v : volume, p : price)

    s = ^CBTP

    print "With security : [s] with "…
    if v > x0 then
            print "Total price : [v * p]"
    else
            print "Volume of zero or negative"
    end if
```

```
end printTotal

function printSecurities()
    Hello("Harry")
    per  security
            print "security : [security] had a last price of : [lastprice]"
    end per

end printSecurities
```

You may call another function inside a function, but remember that the function you are calling must be declared before the function that you are calling from

- Examples of calling the functions :

```
at start
    Hello("Tim")
end start

on trade
    printTotal(security, volume, price)
end on

at end
    printSecurities
end at
```

11.1.2 Things to always remember about user-defined functions

In general, there are a few things that each user must always remember when dealing with user defined functions:

1. Function input parameters and any variables declared inside the functions are local to the functions. The only exceptions are when users use recursive functions (something we hope that none of you need to use), then variables become 'static' variables. This means that each recursive call uses the same variables.
2. You cannot overload user-defined functions in the same way you can overload arrays and built in functions. Each function identifier should be unique and may not be used for any other functions or variables.
3. It is not possible to pass arrays in and out of functions using input parameters and return statements. To use arrays via functions, declare the arrays as global variables. This means that the arrays will be accessible anywhere within the program, in all functions.
4. In a function that returns something, any statements after a return statement are not allowed. This is because the return statement immediately initiates the process of returning something to the function caller, and thus leaving (i.e. terminating) the function. So, any statements after the return statement would have no real use.
5. Both types of user-defined functions may be called directly as a statement. If the function is the type that returns something, then whatever it returns will be ignored.
6. If a function does not have any input parameter then you may call it with the () notation or simply as a standalone word.

11.2 Scope

Scope is an issue that may be very useful to advanced users of Alice that are working on very large and complex Alice files. Large and complex files increase the tendencies of variable name clashes, resulting in incorrect calculations or outputs. You have probably come across the terms *local* and *global* to describe in general the differences between identifiers that are declared in various sections of an Alice file, below are explanations of these terms.

11.2.1 Types Of Scope

Basically there are three different types of scope in Alice, described below.

Scope Types	Description
File scope	Starts at the beginning of the file and ends at the end of the file. This refers to all identifiers that are declared outside of all functions and modules. Variables that have file scope are global, which means that these variables are accessible and usable from within any function, module and when clause in the file.
Module scope	Starts after the module keyword 'module module_name' and ends at the 'end module_name' statement. This refers to identifiers declared within the module body, outside of all the functions and when clauses that is inside the module. Variables of this scope are local to the module it is in. This means that the variables and its values are not accessible or usable by any statements outside of the module.
Function Scope	Similar to Module Scope, it starts after the function keyword 'function function_name' and ends at the 'end function_name' statement. This refers to the variables declared within the function body. Variables of this scope are local to the function it is in, which means that the variables and its values are not accessible or usable outside of the function.

When considering scope, it is also important to always keep in mind that Alice allows the re-declaration of variables and arrays. Which means, values of these kinds of variables and arrays may differ in various places. Please examine carefully the example below:

```
declare str : string
declare counter : zeroed number

function Foo()
        counter = 4
        str = "Audris"
        print "Hello [str]\n"
end Foo

function Foo1()
      print "Hello [str]"
      declare let str = "Tim"
     print "Hello [str]"
end Foo1

at start
        print "[counter]"
        declare counter : zeroed number
        declare str : string

        str = "Hello World"
        Foo
        print "[str]"
        Foo1
        print "[str]"
        print "[counter]"
end at
```

These have file scope, they have a global lifetime. Generally referred to as global variables. It is accessible and usable by all functions, when clauses and modules.

These assignments are using the global variables. The print statement will output "Hello Audris"

The first print statement is using the global variable 'str' and will output "Hello Audris". The declaration will create a local variable 'str' and initialize it to "Tim". The last print statement will use the local variable 'str' and output "Hello Tim".

This will print the value of global variable 'counter', which is currently 0 (still in zeroed state).

Notice that since we don't have a 'when clause scope' these declarations are treated as re-declaration of the already existing global variables (if exists global variables of same identifiers). This means that any use of str and counter variables from here onwards will access the values of the global variables.

Will output the Foo function first. The next print statement will then output the global 'str' which contains "Audris". Next, Foo1 execute and place its output on screen. The next print statement will print the global variable 'str', which will still contain "Audris". The last print statement will output the value of the global counter, which is 4, as it was assigned in the function Foo.

11.2.2 The ‘module’ Feature

The module feature is a method of localising a section of code. By ‘section of code’ we mean for example a complete when clause(s), declared variables, benchmark section, complete functions, etc. The module feature may not and can not be used inside any when clauses, functions, benchmark section, it must exist outside these.
If the following section of code were added to the above code.

```
module MyModule
declare counter : zeroed number
declare str : string

at start
    str = "Carole"
    counter = 3

    Foo
    print "[str]\n"
    Foo1
    print "[str]"
    print "[counter]\n"
end at
end MyModule
```

Module identifier.

Notice how these variables are local to the module. Any use of ‘str’ and ‘counter’ from here onwards, inside the module, will access the values of these variables, not the above global variables. Any accesses of ‘str’ and ‘counter’ outside of this module will access either the global variables or function local variables.

Foo will assign “Audris” and 4 again to the global ‘str’ and global ‘counter’, then it will print “Hello Audris” on screen. Note that the module local ‘str’ and ‘counter’ still contains “Carole” and 3, unaffected by the assignments in the Foo function which uses the global variables. This is clearly evident in the next print statement, which will output “Carole” on screen and not “Audris”. Foo1 will output “Hello Audris” and “Hello Tim” on screen, again, no change. The next print statements will output on screen “Carole” and 3 consecutively, enforcing the point of local module variable uses.

11.3 Including Other Alice files

Sometimes when creating Alice programs it is desirable to use outputs and functions from previous programs. In the past, this meant that you would need to run that previous program first and or cut-paste code sections from previous programs. This could sometimes be a tedious and impractical process. Now, we have the #include command to simplify the process.

The #include command works as you would expect if you are familiar with the C programming language. For example, below we have the command #include “compare.alice” at the top of the new.alice file. This command will make Alice to place the contents of the file compare.alice in place of that command. So whatever code exists in compare.alice may now be accessible and executed from the current Alice file.

To make life even easier, right clicking on the filename that you are including creates a pop-up that allows you to open the include file in a new ALDIT window, or in the current ALDIT window.

In the file compare.alice

```
module old_features
```

Place all of this file code inside a module, so to avoid name clashes when you include this file in another Alice file.

```
declare dd : date

function foo (o_price : price, c_price : price) : string
        if o_price > c_price then
                return “Higher opening price than a closing price.”
        elsif o_price < c_price then
                return “Higher closing price than a opening price.”
        else
                return “No change between opening and closing price.”
        end if
end foo

at start
        declare p1 : price
        declare p2 : price

        printcsv “output.csv”, “SECURITY, DATE, OPEN PRICE, CLOSE PRICE”

        for dd = 12/6/1997; dd <= 17/6/1997; dd = dd + 1 do
                per security
                        p1 = openprice(dd)
                        p2 = closeprice(dd)
                        printcsv “output.csv”, “[security], [dd], [p1], [p2]”
                        print “[security] has [foo(p1,p2)]”
                end per
        end for
end at

end old_features
```

In the file new.alice

```
#include "compare.alice"
```

This command will be replaced by the content of the specified file (i.e. compare.alice). The when clause 'at start' in compare.alice will be executed, and the function 'foo' is now available to new.alice.

```
declare dd : date
p1[security, date] : price
p2[security, date] : price

at dayend
        declare pr1 : price
        declare pr2 : price
        declare let d = 1/1/1999

        read "output.csv", security, date, p1, p2

        printcsv "new.csv", "SECURITY, DATE, OPEN PRICE, CLOSE PRICE"

        per security
                pr1 = openprice(d)
                pr2 = closeprice(d)

                printcsv "new.csv", "[security], [d], pr1, pr2"
                print "\nFor [security]\nOn [d] : [foo(pr1,pr2)]"

                for dd = 12/6/1997; dd <= 17/6/1997; dd = dd + 1 do
                        print "On [dd] : [foo(p1[security,dd], p2[security,dd])]"
                        printcsv
        "new.csv","[security],[p1[security,dd]],[p2[security,dd]]"
                end for
        end per
end at
```

The module feature makes sure you won't have any name clashes when declaring variables. Note here you may declare 'dd' safely, and not worry about it clashing with the global variable 'dd' of compare.alice. See section 3.4 'Scope' for more on the module feature.

Comment [a1]: Page: 1

11.4 Macros

A feature of *Alice* which may help more advanced programmers is the feature of 'macros'. A macro is a frequently-used string of text which can be replaced by a simple word.
The syntax of macros is identical to that of the C language. This covers both macros with parameters and macros without parameters.

NOTE: **#define** is an exception to the normal *Alice* rule that '#' begins a comment. For more about the comment feature, refer to section 9.2.

An example is as follows:

```
#------------------------------------ MACRO DEFINITION ----------------------------
    #define event(E)        on E
    #define text            "[date] [time] [security]"


#---------------------------------- ALERT RULE DEFINITION -------------------------
    event(control)
        print "[text] [controlstatus]"
    end on

    event(trade)
        if security = ^CBTP then
          print "[text] [volume] @ [price] = [value]"
        end if
    end on
```

When the macro is expanded, the *Alert Rule* would be converted to:

```
    on control
        print "[date] [time] [security] [controlstatus]"
    end on

    on trade
        if security = ^CBTP then
          print "[date] [time] [security] [volume] @ [price] = [value]"
        end if
    end on
```

NOTE: If the macro is too long and needs to span over multiple lines, insert a back slash '\' at then end of each line to indicate that the next line is also part of the macro definition.

11.5 The ‘shell’ command

shell	Replacing the old **email** command, this statement allows you to run a UNIX command. Before it will work, users must have permission for TclPrompt (this command grants the same level of access as a TclPrompt). To obtain such permission, you must contact your system administrator or a SMARTS user who has permission and is familiar with the SMARTS admin program. Using this program permissions for a SMARTS user may be altered accordingly. The **shell** command takes one argument, namely a string representing a UNIX command. As always you may evaluate expressions and embed them in a string with [ ] notation. Below is the syntax for the **shell** command: **shell** <string> e.g. Say you’re out and about but want to be aware of any alerts that may occur during your outing. You have an email command called ‘sendsms’ which sends a short string to a certain email account, which will then be sent to your mobile as an sms message. **shell** “sendsms alert_202 on [security] back to office” You should be aware of how UNIX processes characters. Some punctuation characters have special meanings and therefore will be translated during the call to the email command. For example, the following characters will very likely be modified in some way: $ < > | ~ * ? Such characters should either be enclosed in double-quote marks or preceded with a backslash (‘\’) character. To produce double-quotes from an Alice program, use the \” notation. This will tell Alice to put the double-quote into the string, as opposed to interpreting the “ as meaning terminate the string. Before running your program with the **shell** statement you should test the program thoroughly with **print** statements or **printto** statements replacing the **shell** statement. For the above example, it is obvious that you would not want to receive 100 sms messages. SMARTS Pty Ltd takes absolutely no responsibility for bugged Alice programs or even bugs in the Alice interpreter which have unforeseen consequences – conceptually the dataflow with SMARTS is supposed to be strictly one-way. Warning! Do not try to delete a file (eg. shell “rm file.csv”) that is currently being printed to via the printto or printcsv commands. This is because the file(s) involved with printto and printcsv commands inside an alice program is kept open FOR THE DURATION of the program execution. Deleting it will cause any subsequent printto and printcsv commands after the culprit shell delete command, unable to print to the file.

12. HELPFUL CODE EXAMPLES

12.1 Getting prices, or filtered statistics from a point in time

12.1.1 Linear Price Search

One of the most common user defined functions that SMARTS implements during installation projects is a GetPriceAtTime or GetVolumeWithAttributeSinceTime type function. The FAQ explains the intraday sampling technique used by versions of Alice below SMARTS Product 6.2 (releasing in 2007).

To get the exact last traded price at a point in time requires a data structure to record prices. This structure is most likely to be a list. For example

```
Declare Trade_Price[security,number]: price
Declare Trade_Time[security,number]: time
Declare Trades[security]: number
Declare T: number

On trade
        Trades[security] += 1
        T = Trades[security]
        Trade_Price[security,T] = price
        Trade_Time[security,T] = time
End on
```

This structure gives us a list of trades in chronological order and stores the price and time in an array. We can then write a function to loop over this array and retrieve the price at or before time t.

```
Function GetPriceAtTime(ss:security,tt:time): price

        For declare let xx = Trades[ss]; Trade_time[ss,xx] > tt; xx -=1 do

        End for

        If defined(Trade_Price[ss,xx-1]) then
                Return Trade_Price[ss,xx-1]
        End if

        Return closeprice(date-1)
End GetPriceAtTime
```

The above function will loop back over each trade until it reaches the trade just after the desired time tt. The function then checks that there was a trade price for the next trade back in time (xx may have been the first trade of the day) and returns the price if defined. Otherwise it returns the closeprice from the previous day.

12.1.2 Binary Price Search

The drawback with this function is that it is a linear search and if the number of trades for a given security is large then the search cost may be undesireable. It is possible to create a binary search algorithm which will come to the price in less steps than the linear search if the list is long and the time lookback is more than a few minutes.

```
function TradePriceAtTime(ss:security, tt : time) : price
        declare a : number
        declare b : number
        declare m : number

        a = 0
        b = Trades[ss]
        m = floor((tt - START_TIME) / (time - START_TIME) * Trades[ss])
        if Trade_Time[ss,m] < tt then
                a = m
        else b = m
        end if
        for ; a < b - 1;  do
                print "[m] [Trade_time[ss,m]] [a] [trade_time[ss,a]]  [b] [trade_time[ss,b]]"
                m = floor((a + b) / 2)
                if Trade_Time[ss,m] <= tt then
                        a = m
                else b = m
                end if
        end for
        return Trade_Price[ss,a]
end TradePriceAtTime
```

12.1.3 Aggregated Volume with an Attribute (trade type)

The following data structure will store details about trades, the price, time and type, so that a function may loop over the trades and aggregate volume based on the trade type.

```
Declare Trade_Volume[security,number]: price
Declare Trade_Time[security,number]: time
Declare Trade_Type[security,number]: string
Declare Trades[security]: number
Declare T: number

On trade
        Trades[security] += 1
        T = Trades[security]
        Trade_Volume[security,T] = volume
        Trade_Time[security,T] = time
        Trade_Type[security,T] = tradefield("A")*

End on
```

*trade type may be represented by flags, in which case you will need to create a function to extract the flag type

```
Function GetVolumeWithAttributeSinceTime(ss:security, tt:time, type:string): volume
        Declare let Tempvol = x0
        For declare let xx = Trades[ss]; trade_time[xx] >= tt; xx -=1 do
                If  Trade_Type[ss,.xx] = type then
                        Tempvol +=  Trade_Volume[ss,tt]
                End if
        End for
        Return Tempvol
End GetVolumeWithAttributeSinceTime
```

12.2 Determining the last trade of an auction

Some alerts are designed to identify unusual volume in an auction period, or an unusual price movements, and then identify the participant that bought / sold the most during the auction. Of course to do this, we need to know when we have reached the end of the auction process!

```
On trade
        If flags(+Oi) then
                If clearedvol = x0 then
                        # This is the last trade of the auction
                        # Process the alert conditions now
                End if
        End if
End on
```

Clearedvol is a function that tells you after this trade, how much more volume is left overlapping in the orderbook. If that volume is zero, then there are no more trades to execute.

12.3 Breaking up a tokenised string (5.26.05 and above)

You may have stored data in a string separated by a common character. To break this back up, use the strtok function.

```
At start
        Declare let longstring = “1;2;3;4;5;6;7;8;9;10;11;12;13;14;15;16;17”
        Declare shortstring = dummystring

        for shortstring = strtok(longString, “;”); defined(shortstring) ; shortstring = strtok("", “;”) do
                print shortstring
        end for
end at
```

will print
1
2
3
4
…

13. FREQUENTLY ASKED QUESTIONS

13.1.1 When Alerts are displayed, they have big gaps of spaces inside:

- **Alert** *statements* usually cover several lines in the *Alice* program, because the string which is displayed is too big for one line. In this case, the string is allowed to contain newline characters and an indentation, but this will be removed when the alert is printed. However, if you have just used 'tab' or 'space' characters to move to the next line, then it will appear in the Aldit editor as 2 lines but in actual fact is just a continuation of a single line. To fix this, go to the end of the first line and hit <ENTER>.

13.1.2 It says Unknown Statement over a when-clause, eg. "on entord" or "on trade"

- This probably means that you have not ended the previous statement properly.
- Check that you have all the necessary **end on**, **end at** etc.

13.1.3 I don't get any alerts from a rule, no matter how low I set the thresholds.

- This is usually caused by some function being undefined. If a function or value is undefined, then anything that uses it will also be undefined, and any 'if' statement with an undefined condition-expression will do neither the 'then' part, nor the 'else' part. You should write a special rule to test for an expression being undefined before using it, e.g.:

```
if undefined(avdailyval) or value(11:00) > avdailyval then
```

or:

```
if undefined(value(11:00) > avdailyval) then
    alert 888, "TEST", "Debugging alert: avdailyval =    [avdailyval]"
```

13.1.4 It says that a variable is unknown "Declare it or quote it".

- This means that the variable must be declared at the beginning of the Alice program. This must be stated as follows:

```
declare <variable>: <variable type>
```

For example:

```
declare lasttradetime: time
```

13.1.5 I am using the "declare … : zeroed" syntax to reset values, but they are not being cleared

- The **declare … : zeroed <type>** syntax does *not* cause values to be reset, it merely changes how the program deals with elements of the array which it hasn't seen before. (It is a declaration, not an assignment). If you need to reset values back to zero or undefined, then you need to use loops and **let** statements.

13.1.6 Some expression is undefined (it prints a blank cell)

- When a function returns the value 'undefined', it can be for many reasons. For example, all benchmark functions (avdailyvol and largetradeval etc.) will be undefined if a security is trading for the first time ever (or the first time in 60 days).
- Division-by-zero and other mathematical errors will result in 'undefined'.
- You might be using position functions without a current house defined. For example:

```
on trade
    if pos_net_….. > ….
```

will not give you any alerts because the current house is not defined. You must set the current house, the easiest way is to use a **with** statement. For example, instead you could have:

```
on trade
    with buyerh { blah } end with
    with sellerh { blah } end with
end on
```

- It could be that you are doing this:

```
if X then
    { blah }
else if undefined(X) then
    { blah }
end if
```

This will obviously not work, because if X is undefined then it doesn't do either the **then** or the **else** part. You should check if the value is defined before using it, for example:

```
if undefined(X) then
    { blah }
else if X then
    { blah }
end if
```

13.1.7 I get funny calculations/results when I use percents:

- In many cases, percentage values can be mixed with ordinary numbers. For example, you can do the following:

```
if 3 / 4 < 76% then
    alert …
end if
```

- In this case, the 76% is equivalent to a number of 0.76. However, if the program contained the following, then it would probably be considered a bug, (percents are being used in a way which is probably not what the programmer intended):

```
if (3 / 4) * 100 < 76% then
    alert …
end if
```

13.1.8 My program hangs or works very slowly when I compute benchmarks or alerts:

- Check the **when** statements on each of the alerts. If the when statements are made up of an **per house, per security** this may be causing the programme to take a long time to run. For example, the following will be very slow:

```
every 5 minutes
    per security
        per house
            { blah }
        end per
    end per
end every
```

13.1.9 My program runs out of memory or makes the server thrash:

- If your program is using too much memory, you will sometimes get an "out of memory" error, and at other times the server computer will 'thrash' which is a phenomenon whereby all programs on the server suddenly begin operating extremely slowly and the disk drives go mad.
- There are a number of ways to reduce the amount of memory your Alice program uses:

1. First, it will help to know which arrays are consuming most memory. Sprinkle calls to "**print memorystats**" around your program. Each time this is executed, it will output to the screen (or stdout) a list of arrays with the amount of memory each uses.
2. Secondly, you can clear an array at any time with the 'free arrayname[]' command. This will recover all the memory it uses. You can also clear individual entries by assigning undefined quantities to them (see '**dummyvalue**', '**dummynumber**', '**dummystring**' etc.).
3. Thirdly, see if you can eliminate any of the arrays or reduce the number of entries. Maybe you have some redundant information. Maybe you can organise the program to process some intermediate data on the fly and then clear or reuse the arrays used.
4. Fourthly, if many of the entries in a particular array are zero then you should consider declaring that array as 'zeroed …' e.g.:

   ```
   declare tot_sell_agency[security,house] : zeroed value
   ```

   …This way, any time you assign zero to the entry, it will actually remove the memory used by the entry.

13.1.10 Intraday statistics sampling (a.k.a. 5 minute bucketing) :

Alice stores intraday statistics such as total volume, total value, trueprice, bid, ask, liquidity, etc in time interval buckets that is used to provide answers for time based queries such as trueprice(10:04) or trueprice(-5 minutes).

The default time interval (also called sample time) of this bucketing system is 5 minutes but may be changed using a market configuration option called "alice_res" which is a time based configuration (eg. alice_res=00:03:00 will set the bucketing interval for 3 minutes). The lowest possible interval is 10 seconds. Please note that this configuration should not be set without extensive consideration and verification as it affects the alerting system performance.

The starting time of the sampling process is dependant on the market configuration "start_time".

How it works:

Consider the following scenario as an example:

Time	Trueprice
09:55	$10.00

09:58	$10.05
10:00	$10.10
10:03	$10.10
10:04:49	$9.9

Assuming the sample time is 5 minutes, and the start_time configuration is 08:00, the following holds true:

- At 10:01:49, what would the function call trueprice(-5 minutes) give you? Would it be $10.00 from 09:55, $10.05 from 09:58, or $10.10 from 10:00?
  Because the sampling interval is 5 minutes, we need the closest sampling time to from 10:01:49. 10:01:49 – 5 minutes is 09:56:49, and the nearest sampling to 09:56:49 is 09:55, therefore the trueprice function call will give $10.00 from 09:55.
- At 10:03, what would the function call trueprice(-5 minutes) give you? Would it be $10.00 from 09:55, $10.05 from 09:58, or $10.10 from 10:00?
  Because the sampling interval is 5 minutes, we need the closest sampling time to from 10:03. 10:03 – 5 minutes is 09:58, and the nearest sampling to 09:58 is 10:00, therefore the trueprice function call will give $10.10 from 10:00.

13.2 Try to avoid these constructions.

13.2.1 Don't do constructs like this:

```
on trade
    per security
        { blah }
    end per
end on

on entord
    per house
        { blah }
    end per
end on
```

The 'per' statement is a loop. This example is saying to Alice that "each time you get an order, loop over each security, not just the security pertaining to the order and process 'blah'". 99% of the time this is not what the user intended. Just leave out the 'per' statement.

```
on trade
    with ^BHP
        { blah }
    end with
end on
```

This is a similar situation: the clause is executed once for each trade. The current security i.e. 'security' is automatically initialised to the security for this trade. The 'with' statement will change it. Most of the time the user instead intends to say: **on trade if security = ^BHP then** …

13.2.2 My program is getting very large and repetitious:

- If you find yourself constantly copying-and-pasting slabs of code and repeating a lot of code, then a few simple techniques can usually help you to avoid this situation. Firstly, if you are writing a set of alerts with minor variations on each other, then you should make use of temporary variables. For example:

```
if liquid[security] then
    let thresh = largetradeval[security]
    else let thresh = $1000000
end if
if value > thresh then
    alert 101, "LARGE TRADE", "A big one:  [value] > [thresh]!!"
end if
```

- This will eliminate nearly all repetition in the alert rule. You can give different cases the same alert-code or use another temporary variable to give different alert-codes to the different variations.
- Also, note that you can have one body of code executed on multiple types of events. For example:

```
on entord, amdord, delord
    # check market-maker spreads ...
end on
```

- Sometimes you can avoid repeating the same formula in several places by simple storing the results of the calculation in a variable or array which every part of the code can access.

14. APPENDIX A

14.1.1 "on trade" functions

date	time	volume (# units)	price
value ($'s)	security	buyerh (Buyer house)	sellerh (Seller house)
buyert (Buying trader)	sellert (Selling trader)	buyerc (Buying client)	sellerc (Selling client)
orderidask (order id on the sell side)	orderidbid (order id on the buy side)	tradeid (trade id)	transid (transaction id)
flags (a special syntax, not a function: pass it a filter and it returns a boolean).	reporttime (Time the trade was reported to the market.)	findordervol(orderidbid) findordervol(orderidask)	*See also the 'general security functions'*

14.1.2 "on enter" functions

date	time	volume (# units)	price
value ($'s)	security	house	trader
client	buy_or_sell : returns a string "B" or "S"	orderid	bid : best bid price
ask : best ask price	spread : ask – bid	flags (a special syntax, not a function: pass it a filter and it returns a boolean)	transid (transaction id)
	See also the 'general security functions'		

14.1.3 "on amdord" functions

date	time : The time of the amdord	volume (# units): returns the new volume of the order	oldvolume : Returns the old volume of the order
house	entrytime: The time of the corresponding ENTER	value ($'s): returns the new value of the order	oldvalue ($'s): returns the old value of the order
trader	buy_or_sell : returns a string "B" or "S"	price : the new price of the order	oldprice : the old price of the order
client	transid (transaction id)		
flags (a special syntax, not a function: pass it a filter and it returns a boolean)	(In v2.3.0, 'orderid' meant the original id).	orderid : returns the (new) order id. (In some markets, an AMEND changes the order id, more commonly the order id remains the same.)	oldorderid : returns the original order id
security	*See also the 'general security functions'*		

14.1.4 "on delord" functions

date	time	volume (# units)	price
value ($'s)	security	house	trader
client	buy_or_sell : returns a string "B" or "S"	flags (a special syntax, not a function: pass it a filter and it returns a boolean)	transid (transaction id)
entrytime	*See also the 'general security functions'*		

14.1.5 "on deltrade"/"on amdtrade" functions

date	time	volume (# units)	price
value ($'s)	security	buyerh (Buyer house)	sellerh (Seller house)
buyert (Buying trader)	sellert (Selling trader)	buyerc (Buying client)	sellerc (Selling client)
orderidask (order id on the sell side)	orderidbid (order id on the buy side)	tradeid (trade id)	transid (transaction id)
flags (a special syntax, not a function: pass it a filter and it returns a boolean).	*See also the 'general security functions'*		

14.1.6 "on index" functions

index : the index being updated	indexval : the value of the index in the update	indexval(X,time) : previous values	

14.1.7 "on quote" functions

quoteask : sell price quoted	quotebid : buy price quoted	quotevolask	quotevolbid
See also the 'general security functions'			

14.1.8 "on control" functions

security : this is undefined if it's a market or group control	controlstatus : "C"=Closed, "O"=Opening, "N"=Normal (open), "S"=Suspend		

14.1.9 "per order" functions

entrytime	*See also the "on enter" functions*		

14.1.10 "on info" functions

security : returns the current security, i.e. the	infofield(tag): extracts field 'tag' as a string	infofields : returns a string of all fields	infosensitivity : Is the info price sensitive? (returns

security pertaining to this announcement.		(except the full text field) strung together	positive, negative, neutral or undecided)
Infotranstype: Is the announcement new, an amendment or a deletion? (returns AnnfoNew, AnnfoAmend, or AnnfoDelete).			

14.1.11 General security functions: with security, per security, on trade, etc.:

At most points in an Alice program there is a 'current security' defined: whenever you are in a 'with security', 'per security', 'on trade', 'on entord', 'on amdord', 'ondelord', 'on deltrade', 'on quote', sometimes in an 'on control'.
Note that you can only use the functions where the data exists in the underlying database. Very few installations have all the data necessary for all the following functions.

value(date) : turnover value from 'date' to the current time. Only on-market trades	volume(date) : turnover volume from 'date' to the current time. Only on-market trades	tcount(date) : trade count from 'date' to the current time. Only on-market trades	closeprice(date) : Official close price for 'date'.
valueall(date) : As above but with off-market trades too.	volumeall(date) : As above but with off-market trades too.	tcountall(date) : As above but with off-market trades too.	openprice(date) : Official open price for 'date'.
vwap(date) : value weighted average price for this date (on-mkt only)	maxprice(date) : maximum on-mkt trade price for this date	minprice(date) : minimum on-mkt trade price for this date	*(These are all the 'daytot' functions)*
lastprice : last on-mkt trade price	trueprice : last on-mkt trade price adjusted by lower asks & higher bids	mpprice : midpoint price	*(These are the snapshot metrics)*
bid : highest bid price	ask : lowest ask price	spread : ask - bid	
liqdown : ('liquidity down') : the SMARTS order-depth metric for the bid side	liqup : ('liquidity up'): the SMARTS order-depth metric for the ask side	bid(volume) : Best bid price disregarding the first 'volume' units on offer (used to ignore non-serious orders)	ask(volume) : Best ask price disregarding the first 'volume' units (used to ignore non-serious orders)
sampletime0 : The beginning time of the sample interval used for the intraday metrics	ask(time) : the best ask price at a certain time earlier on the current date	bid(time) : the best bid price at a certain time earlier on the current date	mpprice(time) : the midpoint price at a certain time earlier on the current date
sampletime1 : The end time of the sample interval used for the intraday metrics	*These are the 'intraday metrics'*	liqup(time) : liqup at a certain time earlier on the current date	liqup(time) : liqup at a certain time earlier on the current date

volume(time) : turnover volume of on-mkt trades, approx from 'time' to now.	volumeall(time) : turnover volume of all trades, approx from 'time' to now.	value(time) : turnover valuee of on-mkt trades, approx from 'time' to now.	valueeall(time) : turnover value of all trades, approx from 'time' to now.
tcount (time) : number of on-mkt trades, approx from 'time' to now.	tcountall(time) : number of all trades, approx from 'time' to now.		
securityfield(fieldname) : fieldname is a 2-letter string. Get this field from this security. Returns a string.	security**v**field(fieldname) : Like 'securityfield' but returns a number.	controlstatus : The current status (open, suspended etc.). See the codes in the "on control" functions.	rank(date) : The rank of this security by sorting all stocks on turnover over the period 'date' to the current date.
issuedunits : number of units (shares) on issue	issuedate : date issued	marketcap : equals issuedunits * trueprice	
Futures, options, bills & bonds functions:	underlying : the underlying security or instrument	underlyingprice : the price of the underlying instrument	strikeprice
binomial(underlyingprice), binomial(underlyingprice, volatility) : The Black & Scholes pricing model applied to the current security. Pass in the underlying price or the underlying price and the expected volatility	measurevolatility(underlying, number-of-days) : measure historical volatility over this number of calendar days for 'underlying'	impliedv(price, underlyingprice) : Back out the implied volatility from this derivative price and underlying price	maturitydate
eyield : effective yield on a bill/bond	syield : simple yield on a bill/bond	unitfacevalue : the face value of a contract	

15. RUNNING ALICE FROM THE COMMAND-LINE

It is sometimes useful to be able to run Alice programs from the UNIX/Windows command line. Reasons:

- You might want to do a series of runs and so you want to put the calls in a script
- You want to schedule runs using the 'cron' program, e.g. to precompute benchmarks before the start of the next day
- You want to use the Alice 'debug' statement to do interactive debugging.

To do this, use the 'alertserver' program. The 'alertserver' program is the same program called by Almas when you do a calibration run, and the same program called by the startup script at the start of each day for official alerts.
Type 'alertserver' with no arguments to see a 'Usage' message. This gives you a concise explanation of the available command-line arguments.
The compulsory arguments are: what phase, e.g. Alerting, Benchmarking or both; and what database mode, i.e. calibration or official. There are also various optional arguments. Look at the following examples:

alertserver –m asx –AB –f test.alice –calib TEST 20010414

// Run the file 'test.alice', found in the current directory, on just a single day i.e. 14th April 2001.

// Run it in the market 'asx'. Do the alerting phase, and the benchmarking phase if necessary.
// If it already has benchmarks for this Alice program for this date, it doesn't recompute them.

alertserver –m asx –A –f test.alice –l 99999 –calib TEST 20010401 20010430

// Run the file 'test.alice' for the asx market for the whole month of April. A single set of
// benchmarks will be used for all days, calculated up to the last day of March. These
// benchmarks must have been previously computed – otherwise the program will fail with
// an error. Or maybe you don't use the 'benchmarks_below' feature at all in which case
// benchmarks are irrelevant.
// Allow up to 99999 alerts.

alertserver –m asx –AB–official 20010414

// Run the official Alice file for 14th April 2001 and put the results in the official alerts database.

// It will archive any existing alerts you currently have in the official alerts database.

There is another useful program: 'multirun'. Multirun allows you to do lots of runs with a single command:

multirun –m asx –f test.alice –d 20010401-20010430 –calib TEST

// Run the file 'test.alice' over each day in April. Each day is a separate run, i.e. a separate
// alerts directory, named e.g. TEST20010401, TEST20010402 …, with a separate set of
// benchmarks being run. This can be very slow because of the need to recompute benchmarks
// for each day. Note the dash in-between the dates, this is different to 'alertserver'.

multirun –m asx –f test.alice –d 20010401-20010430 –calib TEST3 MTVOL=3
multirun –m asx –f test.alice –d 20010401-20010430 –calib TEST4 MTVOL=4
multirun –m asx –f test.alice –d 20010401-20010430 –calib TEST5 MTVOL=5

// Run the file 'test.alice' over each day in April, in 3 modes. The 3 modes correspond to
// setting userparam 'MTVOL' to the values 3, 4 and 5. In this example we are testing the
// effect of different numbers of days for the 'medium term volume' interval. The variable
// MTVOL has been set up at the beginning of the Alice file as a userparam.

16. APPENDIX B

Following is a list of all arguments which may be passed on to other **SMARTS** applications from **ALMAS**.
This arguments are defined in the Alice program using the **commandline** function in an **alert** *statement*. The syntax of the commandline function is:

commandline = "-<argument> <argument values>, -<argument> <argument values>,…."

where

<argument> = the list of arguments which can be entered depend on the **viewer** specified. By default, the **viewer** is set to **SPREAD**. Below is a complete list of all arguments available for each **SMARTS** application.

<argument values> = must be the same type as the specified <argument>. If there is more than one argument value which is separated by spaces, then this list must be specified between **{** and **}** brackets. Examples of this are illustrated in the examples below.

16.1.1 Arguments which can be passed on from ALMAS to INFORM:

Argument	Version	Example	Description
-m -market	Unix / Win	demo	Set market for editor and viewer.
-s -security	Unix / Win	DAIGF	Set security for editor and viewer.
-securities	Unix / Win	{DAIGF PORI}	Set securities for viewer (and editor if no editor argument is given).
-esecurities	Unix	{DAIGF PORI}	Set securities for editor (and viewer if no viewer argument is given).
-d -date	Unix / Win	19970101	Set date for editor and viewer.
-dates	Unix / Win	{19970121 19970122}	Set date for viewer (and editor if no editor argument is given).
-edate	Unix	19970122	Set date for editor (and viewer if no viewer argument is given).
-infoid	Unix / Win	123	Set required announcement ID.
-iddate	Unix / Win	19970123	Set required announcement date.
-geometry	Unix	600x450	Set viewer window size.
-gemoetry_editor	Unix	450x350	Set editor window size.

16.1.2 Arguments which can be passed on from ALMAS to REPLAY:

Argument	Version	Example	Description
-m -market	Unix / Win	demo	Set market.
-s -security	Unix / Win	DAIGF	Set security or ideal security list.
-d -date	Unix / Win	19970101 +all	Set date.
-t -time	Unix / Win	{10:00 16:00} +trading	Set time range.
-ropt	Unix	+house~client	Set viewing options. To display an option, insert a '+' in front of the option name. To hide an option, insert a '~' in front of the option name. The list of available viewing options are provided in section 16.1.2.1.
-house	Unix / Win	1	Set house.
-trader	Unix / Win	10	Set trader.
-client	Unix / Win	1	Set client.
-filter	Unix Win	+Of	Set transaction flags filter.
-dosearch	Unix	house {house id}	Determines which filters to search on.
-doshown	Unix	house	Determines which filters to use to select orders to show.
-dohighlight	Unix	client trans,price	Determines which filters to use for highlighting.
-sync	Unix	1	Synchronization mode on (1) or off (0) ?
-tcram	Unix	1	Display all price ticks (1) or only active ticks (0) ?
-ofilter	Unix	1	Hide (1) or don't hide (0) flagged orders.
-mantik	Unix	0.01	Tick size.
-stepspeed	Unix	10	Time between steps in milliseconds.
-rmode	Unix / Win	orders	Trade Screen (trades) or Orders Screen (orders) ?

16.1.2.1 List of options available for the '-ropt' argument:

trans	Show transaction at price tick at which it occurs.
ttics	Show an arrow at the price tick which the current transaction is taking place at.
price	Show prices.
best	Show the best bid/ask price.
quotes	Show the dealer quotes.
total	Show order totals at each price tick.
order	Show orders.
bidsasks	Show "BID" or "ASK".
house	Show house entering the order.
trader	Show trader entering the order.
client	Show client entering the order.
volume	Show order volume.
value	Show order value.
oddlots	Show odd-lot portion of orders.
undisclosed	Show undisclosed portion of orders.
flags	Show order transaction flags.
etime	Show order entry time, if available.
id	Show order ID.
eid	Show entered order ID.
actualprice	Show the actual price of an order if not equal to the corresponding price tick.
transactions	Show the Transaction Log Box.
reverse	Reverse the prices from bottom to top.
wrap	Wrap long lines.
interleave	Interleave bids and asks rather than showing them in time order.
swarn	Warn upon search failures.

16.1.3 Arguments which can be passed on from ALMAS to SPREAD:

Argument	Version	Example	Description
-m -market	Unix / Win	demo	Set market.
-s -security	Unix / Win	DAIGF	Set security or ideal security list.
-index	Unix / Win	DOWPEARCY	Set index.
-d -date	Unix / Win	19970101 +all	Set date.
-t -time	Unix / Win	{10:00 16:00} +trading	Set time range.
-client	Unix / Win	1	Set client.
-clients	Unix / Win	{56,21}	Set groups of clients.
-filter	Unix / Win	+Bi	Set transaction flags filter.
-holidays	Unix / Win	yes	Show holidays (Yes) or not (No) ?
-house	Unix / Win	1	Set house.
-houses	Unix / Win	{1 14} 1>12	Set groups of houses.
-imarket	Unix / Win	demo	Set index market.
-omarket	Unix / Win	demo	Set other security market.
-osecurity	Unix / Win	BABI	Other security
-trader	Unix / Win	10	Set trader.
-traders	Unix / Win	1,2,3,4	Set groups of traders.
-weekends	Unix / Win	yes	Show weekends (Yes) or not (No) ?
-dilute	Unix	Yes	Determines whether to dilute prices
-clearoverlays	Unix	Yes	Clear all Overlays?
-logscale	Unix	No	Scale prices with log scale
-maxalerts	Unix	999	Maximum alerts to display
-maxhalts	Unix	32	Maximum halts to display
-maxopen	Unix	100	Maximum market openings/closures to display
-maxtrades	Unix	10,000	Maximum trades to display
-maxtradesize	Unix	1,000,000	Set the maximum trade volume filter (this filter is under the Filter Trade Sizes option)
-maxtradevalue	Unix	1,000,000	Set the maximum trade value filter (this filter is under the Filter Trade Sizes option)
-mintradesize	Unix	1,000	Set the minimum trade volume filter (this filter is under the Filter Trade Sizes option)
-mintradevalue	Unix	100	Set the minimum trade value filter (this filter is under the

Argument	Version	Example	Description
			Filter Trade Sizes option)
-oscale	Unix	Normalise	Set scaling type of other security. Option includes 'none' to not scale the other security true price, 'normalise' to normalise the other security trueprice or 'mean' to scale the other security trueprice by the mean.
-overlays	Unix	{+axes –oxes}	Select the overlays to be displayed. To display an overlay insert a + in front of the overlay name. To hide an option, insert a '-' in front of the overlay name. The list of overlays is provided in section 16.1.3.1
-price	Unix	{1.00 2.00}	Sets the price range. This price range will only be used if the user chose the 'fix' scale type.
-pscale	Unix	Market	Price scaling type. Prices can be scaled based on the 'market', 'alltrades', 'stdev','minmax' or 'fix'
-pwidth	Unix	1.3	Price scaling width
-showalltrades	Unix	Yes	Show all trades (yes) or only those matching filters (no).
-textime	Unix	Yes	Use execution time (yes) or report time(no)

16.1.3.1 List of options available for the '-overlays' argument:

alerts	Alerts
axes	Time/Price Axes
ratioaxes	Ratio Axes
liqtics	Liquidity/Depth Axes
voltics	Value/Volatility Axes
parti	Participant Measures
cap	Papitalization Changes
dilution	Dilution Factor
dots	Time/Price Dots
halts	Trading Halts
holidays	Holidays
inform	Information Events
latency	(Market Order):(Limit Order) Trades
limbalance	Buy:Sell Liquidity Imbalance
liq1	Liquidity
liq100	Orders Depth
midprice	Midpoint Price
oaxes	Other Security Axis
oimbalance	Buy:Sell Depth Imbalance
olayprice	Other Security True Price
open	Market closed
schedule	Orders
trans	Transactions
spread	Bid/Ask Spread
ticks	Price Ticks
trades	Trades
trdprice	Trade Price
truprice	True Price
turnover	Turnover as Percentage of Day
volatility1	Stddev Of Midpoint Price
volatility2	Stddev Of Returns
volume	Time Averaged Volume
value	Time Averaged Value

16.1.4 Arguments which can be passed on from ALMAS to REPORT:

Argument	Version	Example	Description
-m -market	Unix / Win	demo	Set market.
-securities	Unix / Win	DAIGF {DAIGF PORI} +top10	Set the security or list of securities.
-d -date	Unix / Win	19970101 +all	Set date.
-t -time	Unix / Win	{11:00 12:00} +trading	Set time range.
-houses	Unix / Win	1,21 1>21	Set house or houses.
-traders	Unix / Win	13 13,15	Set trader or traders.
-clients	Unix / Win	1	Set client or clients.
-transactions	Unix / Win	TRADE {AMEND DELET}	Set transaction types to be included.
-filter	Unix / Win	+Bi {-Of}	Set transaction flags filter. Note: for negation of filters, flags must be enclosed in curly brackets Eg {-Of} or placed at the very end of the commandline string.
-splittrades	Unix / Win	yes	Determines whether to split trades or not. Options available are '**yes**' to Split trades and '**no**' to not Split trades.
-iorder	Unix / Win	12248308	Set order id numbers to be selected.
-showtrail	Unix	Yes	Determines whether to accumulate order id's or not.
-side	Unix	Bid	Set transactions sieds to be shown, 'bid', 'ask' or '+all'.
-minvalue	Unix	10000	Set minimum value for each transaction.
-maxvalue	Unix	1000000	Set maximum value for each transaction.
-showstats	Unix	Yes	Turn on column stats at base of table 'yes' or 'no'
-updateinterval	Unix	60	Set realtime update interval in seconds

16.1.5 Arguments which can be passed on from ALMAS to COLLUDE:

Argument	Version	Example	Description
-m -market	Unix	demo	Set market.
-s -security	Unix	DAIGF	Set security or ideal security list.
-securities	Unix	{DAIGF PORI} +top10	Set list of securities.
-d -date	Unix	19970101 +all	Set date.
-house	Unix	1	Set single house.
-houses	Unix	1,21	Set pair of houses.
-filter	Unix	+Bi-OF	Set transaction flags filter.
-sort	Unix	turnover	Set type of sorting.
-lastprice	Unix	10.00	Set last price to be used in the calculation of returns.
-retcalc	Unix	wap	Returns calculation type. The types of returns calculation available is wap, lifo or fifo.

17. INDEX