BIGFIX®

BigFix® Relevance Language™ Reference

BigFix, Inc.
Emeryville, CA

Last Modified: November 3, 2006

Version 6.0

Contents

PREFACE 1

AUDIENCE 1
ORGANIZATION OF THIS REFERENCE 2
CONVENTIONS USED IN THIS REFERENCE 2

INTRODUCING THE RELEVANCE LANGUAGE 3

USING THE RELEVANCE DEBUGGER 5

RELEVANCE LANGUAGE OVERVIEW 6

PRIMARY ELEMENTS 6
EXISTS 8
PLURALS (COLLECTIONS) 9
PROPERTIES AND REFERENCES 11
RELATIONS 11
CASTING 13
INDEXING 13
WHOSE - IT 14
TUPLES 19
PLURALS WITH TUPLES 20
ARITHMETIC 21
ANDS AND ORS 22
IF-THEN-ELSE 23
EXPRESSIONS 24

BIGFIX INSPECTORS 25

CORE INSPECTORS 26
OTHER INSPECTOR EXAMPLES 27
PROPAGATION OF NON-EXISTENCE 27
DETERMINING OBJECT PROPERTIES 28

RELEVANCE IN PROPERTY ANALYSIS 30

VIEWING PROPERTY ANALYSES 30
CREATING PROPERTY ANALYSES 30

RELEVANCE IN ACTION SCRIPTS 31

VIEWING ACTION SCRIPTS 31
CREATING ACTION SCRIPTS 32

USEFUL RELEVANCE EXPRESSIONS 34

MANIPULATING STRINGS 34
MANIPULATING DATES AND TIMES 35
COMPARING VERSIONS 38
INSPECTING THE WINDOWS REGISTRY 39
DETERMINING THE OS LANGUAGE 41
RECOGNIZING OFFICE SERVICE PACKS 41
DETECTING FOREIGN LANGUAGE SERVICE PACKS 43
DECONSTRUCTING XML 44
USING WHITE LISTS 45

APPENDIX 47

RELEVANCE LANGUAGE GRAMMAR 47
RELEVANCE OPERATORS 48
PRECEDENCE AND ASSOCIATIVITY 48
RELEVANCE KEY PHRASES 50
LANGUAGE HISTORY 53
ERROR MESSAGES 55

INDEX 60

Preface

The ***BigFix Relevance Language Reference*** is a guide to the **BigFix Relevance Language**™. With this reference and the appropriate ***Inspector Guides***, you will be able to write Relevance expressions that can trigger **Fixlet**® messages.

You can learn how to create Actions by adding the ***BigFix Action Language Reference*** to your reading list. Relevant Fixlet messages, with their associated Actions, are typically deployed to remediate managed computers as well as to inspect the local machine. All the data generated along the way can be tabulated or charted.

Audience

This reference is for IT managers who want to write Fixlet messages for computers managed by the **BigFix Enterprise Suite (BES)**. These Fixlet messages and their associated Actions provide you with great control over your operating environment and provide feedback in near real-time. BES allows you to quickly target and deploy your own custom Fixlet messages to hundreds of thousands of globally networked computers from a centralized console.

This reference is also for IT managers who want to better understand the Relevance expressions that trigger a Fixlet message. Although the language is designed to be human-readable, it is possible to make sophisticated expressions that require careful analysis. The curious will be able to understand the Relevance clauses embedded in Action scripts as well.

Organization of this Reference

This reference is intended to cover the fundamental aspects of the Relevance language and its extended Inspectors. It is broken down into these main sections:

Relevance Language Overview. This section provides an introduction to the Relevance language, discussing the language elements, operators and relations.

BigFix Inspectors. This section shows you how to use Inspectors, which can be thought of as extensions to the Relevance language. Inspectors directly interrogate the client computer and return data that can be logged, output or further evaluated.

Relevance in Property Analysis. This section shows you how to use Inspectors to analyze properties of client computers within the BES Environment, This allows you to chart and report on customized client properties.

Relevance in Action Scripts. This section shows you how to include Relevance expressions in Action scripts, allowing great flexibility in targeting and executing your Actions.

Useful Relevance Expressions. This section discusses some real-world examples.

Appendix. This section includes tables of language elements and grammar, as wells as a list of error messages.

Conventions Used in this Reference

Square bullets and a mono-spaced font denote actual examples of Relevance Expressions. Typically this is in the format of a Q: and A: set, corresponding to the query and the answer:

```
Q: month 9
A: September
```

Introducing the Relevance Language

BigFix allows large networks of computers to be easily monitored and patched in real time using Fixlet® technology. Fixlet messages inspect client computers and report back to relays that in turn report to central servers. This allows patches and updates to be efficiently applied to just those computers that need it, and no others. It also allows the retrieval of various computer properties that can be collected and charted.

The heart of the Fixlet technology is the **Relevance Language** that allows authors to interrogate the hardware and software properties of your managed clients. With the Relevance language, you can write expressions describing virtually any aspect of the client environment. Some Fixlet messages are simply designed to return Relevance information to the servers, but most of them suggest **Actions** that can patch or update the client computer. The Actions, in turn, also take advantage of Relevance expressions.

Fixlet messages and Relevance expressions by themselves can only notify the administrator. Actions, on the other hand, are specifically designed to modify the client, so there is a clear dividing line between a Relevance expression and its associated Action – typically a human is required to deploy the Action. For more information, see the ***BigFix Action Language Reference***.

Dividing the labor in this fashion, using Relevance clauses to benignly inspect the client and Actions to fix them (upon approval), the BigFix applications provide an unprecedented degree of both safety and power.

Relevance expressions are designed to be human-readable. This allows users and administrators to examine them before deploying any associated Actions. The language gives you access to thousands of methods and properties known as BigFix Inspectors. The values returned by the Inspectors can be used for calculations and comparisons, allowing the Fixlet to determine relevance and target a computer for action.

Relevance is a continuous thread winding through all of BES. Some of its more important manifestations include:

- **Evaluating Fixlet relevance.** This is the most common place to find the Relevance language and explains how it got its name. As you'll see in detail below, expressions in this language are designed to trigger if (and only if) the client computer exhibits a particular state – thus the Fixlet message is not displayed unless it's relevant. You can see the Relevance expression behind each Fixlet message in the BES Console: click on a Fixlet message from the list, then look at the **Details** tab. You may see more than one Relevance expression; these are all ANDed together to form the final expression.

- **Displaying properties.** From the BES Console, you can retrieve properties of the BES Client computers. There are some built-in properties, but you can also create your own. To see an example, select an item from the **Analyses** tab, then click on the subsequent **Details** tab. Here you will find named Relevance expressions that are used to interrogate some property of the client computer and return a value. Using the extensive Inspector library, you can create your own customized Relevance expressions to examine properties such as 'computer manufacturer', 'brand of cpu', 'DNS Servers', 'operating system' and virtually anything you care to examine. These properties are derived from the application and combination of thousands of Inspectors, allowing scrutiny of almost any aspect of the computer imaginable.

- **Applying machine-dependent values in Actions.** Actions have their own language, but they can incorporate Relevance clauses that are evaluated at run time. That means that the same powerful set of Inspectors that you can use to target a client can also be used to customize the Action. In this context, Relevance clauses are enclosed in curly brackets {}.

- **Making assertions in Actions.** Boolean Relevance clauses can also be used to govern if-then statements and otherwise control the flow of the Action script. These can be used as assertions at run-time about the validity of some procedure or data. This is commonly used in Fixlet Actions to make sure that a downloaded file has the proper size and hash before proceeding. Assertions make your code safer, more robust and easier to debug.

- **Reporting on BES deployment.** BES Session Inspectors can be used to visualize the state of the BES deployment itself. There are hundreds of Inspectors that can examine BES Fixlets, Actions, computers, users, properties, wizards and more. An extensive set of statistical measures are also provided to help you analyze, report and chart the state of your BES deployment.

Using the Relevance Debugger

BigFix offers a program called the Relevance Debugger that is invaluable for Fixlet authors. To get the most out of this reference guide, you should run the program (RelevanceDebugger.exe) and enter examples as you go along. That will allow you explore variations on the theme and get a much better feel for the Relevance language.

To enter a Relevance expression in the Debugger, simply enter it into the top window and press the Evaluate button. This interface executes one statement at a time, which is the default format. There is another format called QnA, which allows multiple statements. This option is available from the **View** menu. The interface is based on the Debugger's predecessor, a program called QnA.

In this view, you type 'Q:' followed by your query, then press the Evaluate button. The query will be evaluated and the answer(s) will be printed beneath it, preceded by 'A:'. If an error is encountered, it will be printed preceded by 'E:'. (Refer to the Appendix for descriptions of Relevance error messages.)

There are a couple of options in the View Menu to provide more information:

Show Evaluation Time: To analyze performance, select this setting. It will show you the elapsed time of the Relevance execution in microseconds. This is important for creating Fixlet messages that are as responsive as possible.

Show Type Information: You can view the Inspector Type of the returned object by selecting this option. Examining the returned type will help you know how to properly combine your results with more complicated expressions.

This guide presents many examples in the QnA format, to make it easy for you to follow along. Examples are in a Courier font, preceded by a square red bullet. For example:

■
```
Q: names of files of folder "c:/"
A: AUTOEXEC.BAT
A: boot.ini
A: CONFIG.SYS
A: IO.SYS
A: MSDOS.SYS
…
A: whitelist.txt
T: 1.944 ms
I: plural string
```

This relevance snippet returns the names of the files on the C: drive, each preceded by an 'A:'. The time for retrieving this information is 1.944 microseconds and the return type is a plural string.

Relevance Language Overview

The Relevance Language, along with the Inspector extensions, is designed to let you mine your client computers for interesting information, or to see if they need remediation. To illustrate specific Relevance elements, the following sections include examples using the Relevance Debugger (using the QnA view style). If you can, run the program (RelevanceDebugger.exe) and enter the examples as you go along.

Primary Elements

The basic building blocks of the language are numbers, strings and expressions that combine them.

-
  ```
  Q: "hello world"
  A: hello world
  ```

This example outputs a string of characters.

Literal strings like this are parsed for one special character: the percent sign. This is an escape character that encodes for other, non-printable characters, specifically control characters and delete. When a percent sign is found, the encoding expects the next two characters to be hex digits producing a one-byte hex value. That hex value is then added to the internal representation of the string, allowing you incorporate otherwise unavailable characters into a string. Since the percent is used as the escape key, to actually get a percent into a string you must use %25, the hex value of percent.

To convert back to an escaped string for output, characters with a hex value less than 20, greater than 7E or equal to 25 are printed as escaped characters, e.g. %25.

Strings aren’t the only primitives:

-
  ```
  Q: 6000
  A: 6000
  ```

This above example demonstrates an integer. You can also do math:

-
  ```
  Q: (8+3)*6
  A: 66
  ```

Primary elements include parenthetical expressions like (8+3) above. These primary elements can be teased apart as well:

```
Q: substrings separated by "-" of "an-over-hyphenated-string"
A: an
A: over
A: hyphenated
A: string
I: plural substring
```

Note in the example above that four values were returned, not just one. This output is typical of a plural Inspector like ‘substrings’. You can filter this list with a ‘whose’ statement:

```
Q: (substrings separated by " " of "who observed what happened, when and
where?") whose (it contains "w")
A: who
A: what
A: when
A: where?
I: plural substring
```

This example shows two clauses in parentheses. The first parenthetical clause creates a list of words (substrings separated by a space). This ‘whose’ clause contains the primary keyword ‘it’ (discussed in greater detail below), that can stand in for another object – in this case, ‘it’ stands in for each of the individual words, and the expression returns just those words that contain the letter ‘w’. How many of these substrings are there?

```
Q: number of (substrings separated by " " of "who observed what happened, when
and where?") whose (it contains "w")
A: 4
```

This expression shows how you can count up the number of items returned and filtered from a plural Inspector. As these examples show, you can get either singular or plural items back from a Relevance expression. What about no items at all? That’s a subject for the next section.

Exists

Exists is an important keyword that returns TRUE or FALSE based upon the existence of the specified object. This is an important technique that lets you test for existence before you test for a value and possibly incur an error. The keyword has two slightly different typical uses. The first is to determine whether a *singular* object specified by an Inspector exists:

```
Q: exists drive "c:"
A: True
```

```
Q: exists drive "z:"
A: False
```

The above examples test for the existence of the specified objects on the client computer. In these examples, you can see that the client has a drive c:, but not a drive z:. Attempting to find out more about the non-existent drive can generate an error. If you aren't sure about the existence of an object, use the 'exist' keyword before you attempt to examine its properties.

The second usage is to determine whether a *plural* result contains any values:

```
Q: exists (files of folder "c:")
A: True
```

This expression returns TRUE, since files exist on drive c:. Note that using the plural property (files) is a safe way to refer to something that may or may not exist. For instance:

```
Q: file of folders "z:"
E: Singular expression refers to nonexistent object.
```

An error is generated here because there is no drive "z:" on the client computer. If you ask for a plural answer,

```
Q: files of folders "z:"
I: plural file
```

It doesn't give you an answer, but it also doesn't throw an error. Nevertheless, both of these constructs can be examined with the 'exists' keyword without causing an error:

```
Q: exists file of folders "z:"
A: False
```

```
Q: exists files of folders "z:"
A: False
```

You can use existence to determine if two directories have any files in common with an expression like this:

```
Q: exists file (names of files of folder "c:/") of folder "c:/old C"
A: True
```

This expression creates inspector objects for each file in the c:\ folder. It then looks for a file of the same name in the folder c:\old C. It returns true if there are any files with the same name.

Plurals (Collections)

As you saw in the preceding section, plurals of Inspectors are easy to create, typically by adding an ‘s’ to the end of the name. ‘Substring’ is singular, ‘substrings’ is plural:

```
Q: substrings separated by " " of "a short string"
A: a
A: short
A: string
I: plural substring
```

But a plural Inspector doesn’t have to return a plural result:

```
Q: substrings separated by " " whose (it contains "o") of "a short string"
A: short
I: plural substring
```

Although the result is a plural substring type, there is only a single value. In fact, as you saw in the last section, a plural expression can return no value at all, without incurring an error:

```
Q: substrings separated by " " whose (it contains "z") of "a short string"
I: plural substring
```

This returns no values, but no error either. So it’s important to remember that plurality is a property of the expression itself, not necessarily the results.

Furthermore, there are restraints on singular expressions. Whereas a plural can return zero, one or more values, a singular expression is expected to return a single value. For example,

```
Q: substring separated by " " whose (it contains "o") of "a short string"
A: short
I: singular substring
```

You should be expecting a solitary value like this as a result of evaluating a singular inspector. However, the following returns an error:

```
Q: substring separated by " " whose (it contains "s") of "a short string"
E: Singular expression refers to non-unique object.
```

This is because there are two words containing ‘s’, and this expression is looking for a singular value. While two is too much, zero is not enough:

```
Q: substring separated by " " whose (it contains "z") of "a short string"
E: Singular expression refers to nonexistent object.
```

If you’re certain of retrieving a solitary value, use the singular version. Otherwise, for greater flexibility, use the plural. As a practical example, you can find a single folder like this:

```
Q: name of folder of folder "c:/Documents and Settings"
A: All Users
E: Singular expression refers to non-unique object.
```

But as you can see, even though it returns an answer, it also generates an error. This is because there are multiple folders in the specified location, and this command only retrieves the first one. To see the complete list, you need to use the plural version:

```
Q: names of folders of folder "c:/Documents and Settings"
A: All Users
A: Default User
A: LocalService
A: NetworkService …
```

You can explicitly create plurals using a semi-colon (;) to separate the items. These are called collections:

```
Q: "two"; "words"
A: two
A: words
```

```
Q: exist files ("c:\whitelist.txt"; "c:\blacklist.txt")
A: True
```

```
Q: conjunction of (True; True)
A: True
```

```
Q: conjunction of (True; False)
A: False
```

The last two Relevance expressions AND together the semi-colon separated collection. Notice that plurals must be the same type, or you will generate an error:

```
Q: "one"; 1
E: Incompatible types.
```

If you want to combine different types, use a tuple (see below).

Properties and References

Properties of objects can be inspected and referenced. There are thousands of property Inspectors available to cover the majority of software and hardware features of Unix, Windows and Mac systems.

-
  ```
  Q: day_of_week of current date
  A: Tuesday
  ```

Returns a reference to the day of the week from today's system date.

-
  ```
  Q: year of current date
  A: 2006
  ```

Returns the year portion of today's date

-
  ```
  Q: number of processors
  A: 2
  ```

Returns the number of processors in the client computer.

-
  ```
  Q: names of local groups
  A: Administrators
  A: Backup Operators
  A: Guests
  ```

Returns a plural property (names) as a list corresponding to the local group names.

-
  ```
  Q: bit 0 of 5
  A: True
  ```

Returns the zero (low order) bit as True (1) or False (0).

Relations

You use relations to compare values in the Relevance language. There are the standard alpha and numeric comparators (=, !=, <, >, >=, <=) as well a few strictly string relations (starts with, ends with, contains). Here are some examples of expressions that use relations:

-
  ```
  Q: 1 < 2
  A: True
  ```

-
  ```
  Q: 2 is not less than or equal to 1
  A: True
  ```

String compares use alphabetic order:

-
  ```
  Q: "the whole" is greater than "the sum of the parts"
  A: True
  ```

Some relations look for substrings of other strings:

-
  ```
  Q: "nowhere" starts with "now"
  A: True
  ```

-
  ```
  Q: "nowhere" ends with "here"
  A: True
  ```

-
  ```
  Q: "nowhere" contains "her"
  A: True
  ```

-
  ```
  Q: "he" is contained by "nowhere"
  A: True
  ```

Relations return a Boolean TRUE or FALSE depending on the outcome of the comparison. Here is a table of the relation symbols and their English equivalents:

Symbol	English Version
=	is
!=	is not
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
	starts with
	ends with
	contains
	is contained by

Casting

Types can be converted, making it easy to create, concatenate and combine Inspectors into complex Relevance expressions.

-
  ```
  Q: "01 Apr 2020" as date
  A: Wed, 01 Apr 2020
  ```

Converts (casts) a string into a date type.

-
  ```
  Q: 5 as month
  A: May
  ```

Converts an integer into the corresponding month type.

-
  ```
  Q: january as three letters
  A: Jan
  ```

Converts the month January into a three-letter abbreviation.

The Relevance Debugger casts values to strings in order to print them. If an object does not result in a string, the debugger uses the 'as string' method of the object to turn it into a string. If the object can't be cast as a string, an error message is displayed.

Indexing

You can index into lists of objects to select the desired property.

-
  ```
  Q: line 2 of file "c:/frost_poem.txt"
  A: his house is in the village, though.
  ```

Returns the second line of the specified text file.

-
  ```
  Q: month 9
  A: September
  ```

Returns the name of the ninth month.

Whose - It

'Whose' and 'it' are a popular pair in the Relevance language, although 'it' has a life of its own. The following sections detail first the 'whose' and then the 'it', but of necessity, there's a lot of overlap.

Whose

The '**whose'** clause allows you to filter a result or set of results based on specified relevance criteria. It has the form:

<list> whose <filter expression>

For instance:

```
Q: (1;2;3;5;8;17) whose ( it mod 2 = 1 )
A: 1
A: 3
A: 5
A: 17
```

The special keyword '**it**' refers to the elements of the list – in this case the collection of numbers – and is bound only within the filter expression. The Relevance language executes the filter expression once for every value in the filtered property, with 'it' referring to each result in turn. The results where the filter clause evaluates to TRUE are included in the output list. Note that 'it' always refers to the list immediately to the left of the 'whose' statement.

'It' can also refer to direct objects that are not part of a whose clause:

```
Q: (it * 2) of (1;2;3)
A: 2
A: 4
A: 6
```

Here, 'it' takes on the values in the list, one at a time.

You can also use parentheses to define the scope of the whose-it objects. A judicious use of parentheses can ensure proper results while improving readability. For instance, the following examples show how subtle rearrangement of whose clauses can change the output significantly:

```
Q: preceding texts of characters of "banana" whose (it contains "n")
A:
A: b
A: ba
A: ban
A: bana
A: banan
```

```
Q: preceding texts of characters of ("banana" whose (it contains "n"))
A:
A: b
A: ba
A: ban
A: bana
A: banan
```

These expressions both go character-by-character through the word ‘banana’ and return the text preceding each character. Because it returns the text before the character, it returns the blank before ‘b’ and stops at the final ‘a’ with ‘banan’. The expressions both return the same values, but the second one makes it more clear what ‘it’ refers to, namely ‘banana’. Since ‘banana’ will always have an ‘n’, this expression will return *all* the specified substrings.

```
Q: preceding texts of characters whose (it contains "n") of "banana"
A: ba
A: bana
```

```
Q: preceding texts of (characters of "banana") whose (it contains "n")
A: ba
A: bana
```

These two expressions are equivalent, but the second one shows more explicitly what ‘it’ refers to, namely the characters of the word ‘banana’. The ‘n’ appears twice in banana, and so two substrings are returned.

```
Q: preceding texts whose (it contains "n") of characters of "banana"
A: ban
A: bana
A: banan
```

```
Q: (preceding texts of characters of "banana") whose (it contains "n")
A: ban
A: bana
A: banan
```

These two expressions do the same thing, but the second one makes it obvious that 'it' refers to the text preceding the character. Thus only the initial substrings of 'banana' that contain an 'n' are returned.

In practical usage, you could use 'whose-it' clauses to filter folders:

```
Q: names whose (it contains "a") of files of folder "c:"
A: atl70.dll
A: blacklist.txt
A: pagefile.sys…
```

Or you can put the 'whose' clause at the end of the expression, which makes the object of 'it' more explicit and may be easier to read:

```
Q: (names of files of folder "c:") whose (it contains "a")
A: atl70.dll
A: blacklist.txt
A: pagefile.sys
```

If the filtered property is singular, the result of the 'whose' clause is singular. If the filtered property is a plural type, the result is a plural type.

```
Q: exists active device whose (class of it = "Display")
A: True
```

This singular property evaluates to true if there is an active display device on the client computer.

```
Q: files whose (name of it starts with "x") of system folder
A: "xactsrv.dll" "5.1.2600.2180" "Downlevel API Server DLL" "5.1.2600.2180
(xpsp_sp2_rtm.040803-2158)" "Microsoft Corporation"
A: "xcopy.exe" "5.1.2600.2180" "Extended Copy Utility" "5.1.2600.2180
(xpsp_sp2_rtm.040803-2158)" "Microsoft Corporation"
```

This plural expression returns a list of system files whose names start with 'x'.

As it loops through the plural values, the expression in the filter may attempt to evaluate a non-existent object. By itself, such an expression would throw an error such as:

```
E: Singular expression refers to nonexistent object.
```

But in the case of a 'whose' clause, the non-existent value is simply ignored and gets excluded from the resulting set. As a side effect, this feature allows you to examine an object for existence before you attempt to inspect it (and throw an error). As an example, here's a Relevance clause that will trigger an existence error:

```
Q: exists file of folder "z:\bar"
E: Singular expression refers to nonexistent object.
```

But, by placing this clause inside a 'whose' statement, you can avoid the error:

```
Q: exists true whose ( exists file of folder "z:\bar" )
A: False
```

It

The ‘it’ keyword always refers to the closest direct object or the object of the closest enclosing ‘whose’ clause, whichever is closer. There are three simple contexts in which ‘it’ has a meaning:

- <‘it’ expression> of <direct_object>
- phrase (<‘it’ expression>) of <direct_object>
- (<whose_object>) whose (<‘it’ expression>)

The first two contexts involve direct objects, the third involves a ‘whose’ clause. An example of a direct object is this expression, which lists the names and file sizes of a specified folder:

```
Q: (name of it, size of it) of files of folder "c:"
A: AUTOEXEC.BAT, 0
A: blacklist.txt, 42
A: boot.ini, 209
A: CONFIG.SYS, 0
…
A: whitelist.txt, 213
```

Here, ‘it’ refers to the ‘files of folder “c:”’.

The ‘whose’ clause lets you filter a list based on the evaluation of an ‘it’ expression. This is one of the most important targets of the ‘it’ keyword:

```
Q: exist files whose (name of it starts with "b") of folder "c:"
A: True
```

```
Q: number of (files whose (name of it starts with "b") of folder "c:")
A: 2
```

In these expressions, ‘it’ still refers to the ‘files of folder “c:”’.

You must be careful about the placement of parentheses, which can change the target of the ‘it’ keyword. In the following expression, ‘it’ refers to files:

```
Q: (files of folder "c:") whose (name of it contains "a")
A: "atl70.dll" "7.0.9466.0" "ATL Module for Windows (Unicode)" "7.00.9466.0"
"Microsoft Corporation"
A: "blacklist.txt" "" "" "" ""
…
```

Note that this is not the same as the following Relevance expressions, which both have the wrong placement of parentheses:

```
Q: files of  folder "c:" whose (name of it contains "a")
E: Singular expression refers to nonexistent object.
```

```
Q: files of ( folder "c:" whose (name of it contains "a") )
E: Singular expression refers to nonexistent object.
```

These are two equivalent (and wrong) statements where the ‘it’ refers to the closest object, which is the folder, not the files.

There can be more than one ‘it’ in an expression. The rule is that each one refers to the objects listed to the left of the associated ‘whose’. For instance:

```
Q: preceding texts whose (it contains "n") of characters whose (it is "a") of
"banana"
A: ban
A: banan
```

Here the expression returns the substrings preceding ‘a’ that contain ‘n’. The first ‘it’ refers to the substrings; the second refers to the characters. This simple and intuitive rule makes it easy to develop complex expressions. Here’s another example:

```
Q: (characters of "banana") whose (exists character whose (it is "n") of
preceding text of it)
A: a
A: n
A: a
```

This expression illustrates two nested whose-it clauses. The inner one finds leading substrings that contain an ‘n’. The outer one returns the characters following those substrings.

Since ‘it’ represents a value, you can operate on it like any other variable:

```
Q: (it * it) of (1;2;3;4)
A: 1
A: 4
A: 9
A: 16
```

You can nest these references:

```
Q: (it * it) whose (it > 8) of (1;2;3;4)
A: 9
A: 16
```

Here, the first instances of ‘it’ are multiplied and passed on to the third instance of ‘it’ for comparison.

‘It’ never stands for an expression, but rather for a single value. Often it stands, in turn, for the serial values of a plural expression. But it can only stand for one value at a time.

Tuples

Tuples add some useful properties to the Relevance language. A Tuple is basically a compound type composed of two or more other types. It can be returned directly from an Inspector, like this:

```
Q: (now & (1*hour)) * true
A: ( Fri, 22 Sep 2006 15:25:43 -0400 to Fri, 22 Sep 2006 16:25:43 -0400 ), True
I: timed( time range, boolean )
```

This Relevance clause returns a compound object including a time range and an associated Boolean TRUE/FALSE. Notice the use of the concatenation operator (&), used here to create a time range (see arithmetic operators, below).

Tuples can also be explicitly generated using the comma (**,**) keyword. Any mix of types is allowed:

```
Q: number of processors, "B or not", 8/4, character 66
A: 2, B or not, 2, B
I: ( integer, string, integer, string )
```

```
Q: now, "is the time"
A: ( Fri, 22 Sep 2006 12:14:55 -0400 ), is the time
I: ( time, string )
```

```
Q: 1, number of processors < 3, "friend"
A: 1, True, friend
I: ( integer, boolean, string )
```

Note that if an individual Inspector returns a tuple, it will always return the same types in the same order. It's not possible to have an Inspector return tuples of type <int, string, int> in one case and <int, int, string> in another.

Tuples can also be indexed by using the '**item**' keyword (indices start at 0). For instance:

```
Q: item 0 of ("foo", 3, free space of drive of system folder)
A: foo
I: singular string
```

```
Q: (item 1 of it; item 2 of it) of ("foo", 3, free space of drive of system
folder)
A: 3
A: 18105667584
I: plural integer
```

Tuples provide a way for a relevance expression to return several related properties. For instance, you could generate a set of filenames and corresponding file sizes for all files that meet a specific criteria with a Relevance statement like this:

```
Q: (name of it, size of it) of files whose ( size of it > 100000 ) of folder
"c:"
A: hiberfil.sys, 536301568
A: ntldr, 250032
A: pagefile.sys, 805306368
I: plural ( string, integer )
```

Plurals with Tuples

Tuples can be combined with plurals to create Relevance clauses of surprising complexity and power. The easiest combination is also the least useful. Forming plurals of tuples (of the same type) just creates a plural tuple:

```
Q: (1,2); (3,4)
A: 1, 2
A: 3, 4
I: ( integer, integer )
```

However, attempting to form a plural of tuples of *different* types yields an error. As we've already seen, plurals must always be of the same type:

```
Q: (1,2);("a","b")
E: Incompatible types.
```

Interestingly, forming a tuple of plural expressions generates a set of tuples that represents the cross product of all the component plurals:

```
Q: ((1; 2), ("a"; "b"), ("*"; "$" ))
A: 1, a, *
A: 1, a, $
A: 1, b, *
A: 1, b, $
A: 2, a, *
A: 2, a, $
A: 2, b, *
A: 2, b, $
I: plural ( integer, string, string )
```

Tuples of plurals can also be used to search two lists for commonality. For example, suppose we have two lists of integers, and want to know what numbers are in the intersection of the lists. We can do this by using a nested whose, and then we refer to the outer list by wrapping it in a tuple:

```
Q: (1;2;3;4) whose (exists (it, (2;4;6;8)) whose (item 0 of it is item 1 of it))
A: 2
A: 4
```

The downside of this method is that the second list is bound within the ‘whose’ clause and must be recreated for every iteration. To maintain responsiveness, you should keep lists like this short.

Tuples of plurals can also be used to compare two sets of data:

```
q:((1;2;3;4),(5;6;7;8)) whose (item 1 of it = 2*item 0 of it)
A: 3, 6
A: 4, 8
```

You can also find out just which files are in common by serially comparing the tuples of ‘new folder, old folder’:

```
Q: (names of files of folder "c:/") whose (exists (it, (names of files of folder
"c:/old C")) whose (item 0 of it is item 1 of it))
A: CONFIG.SYS
A: IO.SYS
A: MSDOS.SYS
A: report.txt
```

Arithmetic

The Relevance language includes the typical binary mathematical functions, including addition, subtraction, multiplication, division and modulo.

```
Q: 21 mod 5
A: 1
```

Returns the integer corresponding to 21 modulo 5.

```
Q: 36*month/2
A: 1 years, 6 months
```

Multiplies and divides months resulting in a ‘month and year’ type.

```
Q: 2+3
A: 5
```

Adds integers together to produce a sum.

```
Q: current month + 2*month
A: November
```

Adds two months to the current month (in this case, September).

```
Q: december - current month
A: 3 months
```

Subtracts the current month (1-12) from December (12) to produce a ‘number of months’ type.

A few operators in the language are unary (requiring only one argument), such as negation:

```
Q: -(3*5)
A: -15
```

As expected, this minus sign negates its argument (the product in parentheses).

There is another “arithmetic” symbol, the ampersand (&). This is the concatenation operator that joins strings:

```
Q: "now" & "then"
A: nowthen
```

It’s also used to create time ranges:

```
Q: now & day
A: Sat, 21 Oct 2006 21:55:28 -0400 to Sun, 22 Oct 2006 21:55:28 -0400
```

ANDs and ORs

Logical ANDing and ORing are also available as binary operators.

```
Q: version of regapp "wordpad.exe" as string = "5.1.2600.2180" and name of
operating system = "WinXP"
A: True
```

Returns TRUE only if both equations are true (AND expression).

```
Q: name of operating system = "WinNT" or name of operating system = "WinXP"
A: True
```

Returns TRUE if one OR the other equation is true. You can also logically negate a Boolean expression with the ‘not’ keyword.

```
Q: not exists drive "z:"
A: True
```

Returns True is the z: drive doesn’t exist. This is a unary operation (not) being used to negate another unary operator (exists).

If-then-else

If-then-else clauses have the form:

- if <conditional-expression> then <expression1> else <expression2>

Both <expression1> and <expression2> must have the same type, and <conditional-expression> must be a singular Boolean.

If <conditional-expression> is true, then <expression1> is evaluated and returned; otherwise <expression2> is evaluated and returned.

Starting with version 5.1 of BES, if-then-else clauses have been implemented as late-binding, so potential vocabulary errors on the branch not taken are ignored. This makes it safe to write cross-platform Relevance expressions without worrying about throwing errors for incorrect OS-specific Inspectors. For instance, you can write:

```
Q: if name of operating system contains "Win" then name of application
"conf.exe" of registry else "conf.exe"
A: conf.exe
I: singular string
```

On a non-Windows OS, this expression will execute the ‘else’ expression and avoid an attempt to inspect a non-existent registry.

Note: Prior to version 5.1 of BES, both branches were checked to be sure they were meaningful, which could generate an error. In that case, a parse error would have occurred on any non-Windows system when the unknown keyword ‘registry’ was encountered.

If-then statements can be useful for reporting user-defined errors:

```
Q: if (year of current date as integer < 2006) then "Still good" else error
"Expired"
E: User-defined error: Expired
```

This expression throws a user-defined error if the argument is false.

```
Q: if (name of operating system = "WinXP") then "wired" else if (name of
operating system ="WinNT")  then "tired" else "expired"
A: wired
```

This expression does a three-way test of the operating system.

Expressions

Putting all the pieces together yields complete relevance expressions. They can be short and to the point:

-
  ```
  Q: number of active devices
  A: 156
  ```

or they can be extremely specific:

-
  ```
  Q: exists key whose (value "DisplayVersion" of it as string as version >=
  "10.0.6626.0" as version AND (character 1 of it = "9" AND (character 2 of it =
  "0" OR character 2 of it = "1") AND (first 2 of following text of first 3 of it
  = "11" OR first 2 of following text of first 3 of it = "12" OR first 2 of
  following text of first 3 of it = "13" OR first 2 of following text of first 3
  of it = "28") AND (preceding text of first "}" of it ends with "6000-11D3-8CFE-
  0050048383C9")) of name of it) of key
  "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall" of
  registry
  ```

Relevance expressions allow you to analyze and report on specific properties of your client computers with minimal disturbance, so that you act only on the computers that need assistance and none of the others.

BigFix Inspectors

The Relevance language is dedicated to manipulating Inspector objects, which can be thought of as modular extensions of the language. Inspectors are designed to interrogate the software, firmware and hardware of each of the client computers on the network. There are thousands of Inspectors that you can use to analyze various aspects of the computer to make sure that any Actions you propose are properly targeted. Inspectors are also used to produce substituted variables in Action scripts There are OS-specific Inspector libraries for Windows, HP-Unix, AIX, Linux, Solaris and the Macintosh. For more information, see the ***Inspector Library*** for the OS you're interested in.

Many of the keywords of the language are not unique; they get their meaning from their context. An Inspector's context is dictated by the form of the Inspector. There are seven forms:

Form	Syntax required
Cast	<object> as keyword
Global	keyword
Named	keyword "*name*" of <object>
NamedGlobal	keyword "*name*"
Numbered	keyword *number* of <object>
NumberedGlobal	keyword *number*
Plain	keyword of <object>

These differ from one another in format and syntax. Except for Cast, these forms can be used to access both single objects and lists of objects by using the plural form of the keyword.

Core Inspectors

A few basic Inspectors serve to expand the core language. They are similar to Relevance language elements, but they are often OS-dependent and therefore easier to compile into Inspector libraries.

-
  ```
  Q: floating point "3.14159"
  A: 3.14159
  I: floating point
  ```

Creates a floating point number out of the specified string.

-
  ```
  Q: string "hello"
  A: hello
  I: string
  ```

Creates a string type from the specified quoted string.

-
  ```
  Q: nan of (floating point "1.e-99999" / 0)
  A: True
  ```

Nan (Not A Number) is used to test floating point numbers.

-
  ```
  Q: first 6 of "Now is the time"
  A: Now is
  ```

Returns the first N characters of the specified string.

-
  ```
  Q: multiplicities of unique values of (1;2;3;3)
  A: 1
  A: 1
  A: 2
  ```

The multiplicity Inspector allows you to analyze the frequencies of items in a list.

Other Inspector Examples

Some more examples of basic Inspectors include the following:

-
  ```
  Q: now
  A: Thu, 21 Sep 2006 19:39:33 -0400
  I: time
  ```

The 'now' Inspector returns the current day, date, time and time zone from the client computer. This is an Inspector of the 'world' (the environment on the local client computer) that returns a time.

-
  ```
  Q: exists file "c:/report.txt"
  A: True
  I: boolean
  ```

This Relevance expression returns True if the specified file exists. This is a filesystem Inspector that evaluates to Boolean using the existence operator.

-
  ```
  Q: names of folders of folder "c:/program files"
  A: Adobe
  A: BigFix Enterprise
  A: Cisco Systems
  A: iPod
  A: Macromedia
  A: Microsoft Office …
  I: string
  ```

The above expression returns a list of folders that reside in the specified folder. The names of the folders are string types.

-
  ```
  Q: name of current user
  A: John
  I: string
  ```

This phrase returns the name of the current user as a sting type.

Propagation of non-existence

If a property doesn't exist, any other derivatives of that property also don't exist:

-
  ```
  Q: exists folder "z:/foo"
  A: False
  ```

-
  ```
  Q: files of folder "z:/foo"
  E: Singular expression refers to nonexistent object.
  ```

-
  ```
  Q: line 1 of files of folder "z:/foo"
  E: Singular expression refers to nonexistent object.
  ```

Since the original folder doesn't exist, any references to the folder are also nonexistent.

Determining Object Properties

The relevance language has some features built into it that help you to determine the object properties you can query. For example, suppose we evaluate the following relevance expression:

```
Q: folder "c:\"
E: The operator "string" is not defined.
```

This error message means that the relevance expression evaluated successfully, but the relevance language doesn’t know how to display a folder. In order to get some information out of the folder, we’re going to have to query its properties. To do this, we can use the relevance language ‘Introspectors’. The Introspectors return information about the Inspectors currently being used by the relevance debugger and QnA. They contain all the information about what properties of an object can be queried. In essence, they are Inspector Inspectors. For example, to find out the properties of a folder, we use the query:

```
Q: properties of type "folder"
A: descendants of <folder>: file
A: file <string> of <folder>: file
A: folder <string> of <folder>: folder
A: application <string> of <folder>: application
A: files of <folder>: file
A: find files <string> of <folder>: file
A: folders of <folder>: folder
A: security descriptor of <folder>: security descriptor
```

However, this is not an exhaustive list of folder properties. A folder type also has a parent type, the filesystem object type. We can query all the properties of a filesystem object as well. For example, pathname is a property of a filesystem object, but it didn’t show up in the properties query above. However, since folder is a subtype of a filesystem object, we can query the pathname of a folder:

```
Q: pathname of folder "c:\"
A: c:
```

In order to find out whether the folder type has a parent type, use the following relevance query:

```
Q: parent of type "folder"
A: filesystem object
```

Most types do not have a parent type. For example, filesystem object types don’t have a parent type.

```
Q: parent of type "filesystem object"
A: Singular expression refers to nonexistent object.
```

Thus, all of the properties that can be queried of a folder are either properties of folder or filesystem, and so the following relevance expression will list both:

```
■ Q: properties of type "folder"; properties of type "filesystem object"
  A: descendants of <folder>: file
  A: file <string> of <folder>: file
  A: folder <string> of <folder>: folder
  ...
  A: normal of <filesystem object>: boolean
  A: temporary of <filesystem object>: boolean
  A: compressed of <filesystem object>: boolean
  A: offline of <filesystem object>: boolean
  ...
```

An even more thorough list of properties can be discovered using the following expression:

```
■ Q: properties whose ( it as string contains "folder" )
  A: ancestors of <filesystem object>: folder
  A: descendants of <folder>: file
  A: parent folder of <filesystem object>: folder
  ...
  A: application folder <string> of <registry>: folder
  A: application folder of <registry key>: folder
  A: application folder <string> of <registry key>: folder
  A: install folder <integer>: folder
```

Relevance in Property Analysis

Viewing Property Analyses

From the BES Console, click on an item from the **Analyses** tab. In the bottom window, click on the **Details** tab. Here you can see the Relevance expressions behind a property analysis.

For example, select **BES Component Versions** from the Analyses list. Click on the Details tab to see the Relevance expressions behind each Analysis. These Analyses return the client, relay, console and server version of each client computer. For instance, BES Relay Version has the following Relevance statement:

- `if (exists regapp "BESRelay.exe") then version of regapp "BESRelay.exe" as string else "Not Installed"`

This returns the version of the BES Relay, after first determining that it exists. If it does not, it returns 'Not Installed'.

A property can return more than a single item. It can, for instance, return a tuple:

- `(total run count of it, first start time of it, last time seen of it, total duration of it) of application usage summaries "excel.exe"`

This Relevance clause returns several properties that summarize the client's usage of Excel.

Creating Property Analyses

You can create your own properties. These allow you to track any combination of software, hardware and firmware that you desire, across your entire network. Once created, the results of the property analysis can be printed or charted.

For instance, you might want to monitor the status of the operating system languages across your network. You could use a Relevance clause like this to retrieve the information:

```
Q: system language
A: English (United States)
I: singular string
```

You can give this expression a name that you can track, such as 'System Language'. Here's how:

From the **Tools** menu, select **Create New Analysis**. Enter the title and description of the analysis, and then click on the **Properties** tab. Enter your desired Name, Relevance expression and evaluation period in the appropriate fields. Once activated, this Analysis will report back to the BES Server, allowing you to view or chart the results.

Relevance in Action Scripts

Viewing Action Scripts

You can view an Action script from the BES Console by selecting a Fixlet or Task and clicking on the **Details** tab. For more information on the Action syntax, see the ***BigFix Action Language Reference.***

In many of the Action scripts, you can see Relevance expressions inside of curly brackets {}. When the Action is executed, these expressions are evaluated on each client computer and the results are substituted into the Action script. This allows an author to create Actions that are custom-tailored for each client. For instance:

- `run "{pathname of regapp "excel.exe"}"`

This example can run a program without knowing where it is located. The bracketed relevance expression evaluates the pathname automatically using the ‘regapp’ inspector. Embedding Relevance expressions lets you execute precisely targeted Action scripts. This script may use a different pathname on every client, but still operate as intended. This allows you to write readable, compact scripts that will automatically customize themselves to each client on your network.

As well as substituting variables, you can use Relevance expressions to make assertions that can alter the flow of the code:

- `pause while {exists running application "c:\updater.exe"}`

This action pauses until a program finishes executing, using the ‘running application’ inspector.

Substitution is not recursive, although any particular command may have one or more expressions to evaluate before execution. The BigFix application is expecting to find a single expression inside the curly braces. If it sees another left brace before it encounters a closing right brace, it treats it as an ordinary character:

- `echo {"a left brace: {"}`

would send this string to output:

- `a left brace: {`

Therefore no special escape characters are necessary to represent a left brace. To output a literal right brace without ending the substitution, use a double character:

- `echo {"{a string inside braces}}"}`

would send this string to output:

- `{a string inside braces}`

Or consider this example:

- `appendfile {{ name of operating system } {name of operating system}`

When this example is parsed, the double left braces indicate that what follows is *not* a relevance expression. Therefore, this part of the script is treated as a string, up to the first right brace. The third left brace indicates the actual start of a Relevance expression. This outputs the following line to __appendfile:

- `{ name of operating system } WinXP`

Creating Action Scripts

You can create your own Action scripts in the BES Console by selecting **Take Custom Action** from the **Tools** menu. This brings up a dialog box where you can set Relevance, the message and more. Click on the **Action Script** tab. In the text window that shows up, you can enter any Action script you please. You can embed the results of a Relevance expression anywhere in your script by enclosing it in curly brackets {}.

For instance, you might want to download a file with a command like:

-
  ```
  download http://download.bigfix.com/download/bes/60/BESServerUpgrade-
  6.0.12.5.exe
  ```

Then, to make sure it downloaded properly with a secure hash, you could add this command with an embedded Relevance clause:

-
  ```
  continue if {(size of it = 18455939 AND sha1 of it =
  "58a879f5b98357c4ec6f5ff7dbe3307eef5ca2ec") of file "BESServerUpgrade-
  6.0.12.5.exe" of folder "__Download"}
  ```

This expression compares the length of the file (found by looking in the __Download folder) to a known size. It also compares the sha1 of the file to a known value. This construct allows you to stop execution of the Action script if the file was not downloaded properly. This illustrates a common usage of the Relevance language to make an assertion in an action script.

You might want to set a registry key with the results of a Relevance expression:

-
  ```
  regset
  "{"[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\SharedAccess\Parameters
  \FirewallPolicy\" & (if (current profile type of firewall = domain firewall
  profile type) then ("DomainProfile") else ("StandardProfile")) & "]"}"
  "EnableFirewall"=dword:00000000
  ```

Here, the result of the Relevance expression in the curly brackets is substituted into the name of the registry setting. This example shows how to branch based on the value of an Inspector in order to set a registry with the proper string.

You can also set Action variables with the results of Relevance statements. This is done with the parameter command:

- `parameter "tmpfolder" = "{pathname of folder (value of variable "tmp" of environment)}"`

Because the Relevance expression used to target the Fixlet is often the same one used in the corresponding Action, you are more likely to be solving the right problem. That makes script-writing easier and makes scripts more robust and accurate.

Useful Relevance Expressions

This section contains some real-world relevance clauses that are easy to customize for your own use. In these samples, you can see how you might build up a complex expression from some simple, basic elements.

Manipulating Strings

Creating Multiple Results

Multiple substrings can be extracted from a string with commas (,) as delimiters:

```
Q: substrings separated by ", "  of "apple, orange, pear, kiwi"
A: apple
A: orange
A: pear
A: kiwi
I: plural substring
```

Or if you want, just use spaces:

```
Q: substrings separated by " "  of "apple orange pear kiwi"
```

Reversing a String

If you know the length of the string, you can explicitly reverse the order of the characters:

```
Q: concatenation of characters (4; 3; 2; 1; 0) of "abcde"
A: edcba
```

This uses the positions of the characters in reverse order to flip the string. But what if you don't know how many characters are in the string? There are some properties of strings that can be turned to the task:

```
Q: positions of "abcde"
A: 0
A: 1
A: 2
A: 3
A: 4
A: 5
```

This states that there are six positions in the string (including the pointer to the end of the string), corresponding to the number of characters, plus one. As you scan through the string, there are fewer and fewer characters after the specified position:

```
Q: following texts of (positions of "abcde")
A: abcde
A: bcde
A: cde
A: de
A: e
A:
```

The length of these strings can be measured, to produce a list of numbers that is a perfect inversion of the positions listed above.

```
Q: lengths of (following texts of (positions of "abcde"))
A: 5
A: 4
A: 3
A: 2
A: 1
A: 0
```

This inverted list can be used to scan in reverse order through the string, concatenating as you go:

```
Q: concatenation of characters (lengths of (following texts of (positions of
it))) of "abcde"
A: edcba
```

Manipulating Dates and Times

Converting time to mm/dd/yyyy

To convert the time format returned by the “now” Inspector into mm/dd/yyyy format, you extract the three components (month, day, year) and then concatenate them with slashes. Start with the *date* portion of now (excludes the time portion):

```
Q: date (local time zone) of now
A: Mon, 25 Sep 2006
I: date
```

This returns a date with the elements we want to rearrange. The “month” Inspector can give us a properly formatted numeric month (two-digits, with a leading zero):

```
Q: (month of date (local time zone) of now) as two digits
A: 09
I: string
```

The “day_of_month” Inspector returns the date, which we format as above:

```
Q: day_of_month of date (local time zone) of now as two digits
A: 25
I: day of month
```

The year Inspector rounds things out:

```
Q: year of date (local time zone) of now as string
A: 2006
I: string
```

Concatenate these components with slashes to finish it off:

```
Q: (month of date (local time zone) of now) as two digits & "/" & day_of_month
of date (local time zone) of now as two digits & "/" & year of date (local time
zone) of now as string
A: 09/25/2006
T: 0.263 ms
I: string
```

This can be improved considerably by calling the “now” function only once and referring to it elsewhere with the keyword “it”:

```
Q: (month of it as two digits & "/" & day_of_month of it as two digits & "/" &
year of it as string) of date (local time zone) of now
A: 09/25/2006
T: 0.170 ms
I: string
```

This version is shorter, easier to read and about a third faster. Perhaps of greater importance, the value of now can change between invocations, so you may actually get a wrong answer with the first technique. On New Year’s Eve, for instance, you might get December coupled with the wrong year.

Converting yyyymmdd to date

Converting from yyyymmdd to a standard date format uses a different set of Inspectors. First, break the string apart into day, month and year parts:

```
Q: first 2 of following text of position 6 of "20071201" as integer
A: 1
```

```
Q: first 2 of following text of position 4 of "20071201" as integer
A: 12
```

```
Q: first 4 of "20071201" as integer
A: 2007
```

Then convert these integers to their component date types:

```
Q: day_of_month (first 2 of following text of position 6 of "20071201")
A: 1
I: day of month
```

```
Q: month (first 2 of following text of position 4 of "20071201" as integer)
A: December
I: month
```

```
Q: year (first 4 of "20071201" as integer)
A: 2007
I: year
```

These components are then concatenated to produce a standard date:

```
Q: day_of_month (first 2 of following text of position 6 of "20071201" as
integer) & month (first 2 of following text of position 4 of "20071201" as
integer) &  year (first 4 of "20071201" as integer)
A: Sat, 01 Dec 2007
I: date
```

This can be simplified by using the ‘it’ keyword as a variable representing `"20071201"`:

```
Q: (day_of_month (first 2 of following text of position 6 of it as integer) &
month (first 2 of following text of position 4 of it as integer) &  year (first
4 of it as integer)) of "20071201"
A: Sat, 01 Dec 2007
I: date
```

A similar result can be accomplished by using a regular expression. The date can be extracted by choosing the first four digits:

```
Q: parenthesized part 1 of ( matches (regex "(\d\d\d\d)(\d\d)(\d\d)" ) of
"20051201")
A: 2005
```

The various date segments can be assembled along these lines to create:

```
Q: (day_of_month (parenthesized part 3 of it as integer) & month (parenthesized
part 2 of it as integer) & year (parenthesized part 1 of it as integer))of
(matches (regex "(\d\d\d\d)(\d\d)(\d\d)") of "20051201")
A: Thu, 01 Dec 2005
```

As above, this expression uses the ‘day_of_month’ Inspector to return a date corresponding to the concatenation of the components.

Comparing Versions

Numeric version comparisons can be tricky, because they are not numbers in the traditional sense. Version numbers typically have multiple segments separated by periods, such as “6.01.2.3”. A common (but not universal) structure numbers the releases like this:

major.minor[.revision[.build]]

So, when you compare versions, you need to specify all the relevant segments to get a proper comparison. If you compare them as if they were integers or floating point numbers, you may get the wrong answer. Consider these examples:

-
  ```
  Q: "6" as version < "6.44" as version
  A: False
  ```

-
  ```
  Q: "6.0" as version < "6.44" as version
  A: True
  ```

The second relevance expression works, because it has the same number of version segments, so it compares properly.

-
  ```
  Q: "5" as version = "5.50" as version
  A: True
  ```

-
  ```
  Q: "5.00" as version = "5.50" as version
  A: False
  ```

The second expression fails properly, because it compares a two-segment version to another two-segment version.

Don’t assume the version segments are two-digits:

-
  ```
  Q: "5.100" as version > "5.99" as version
  A: True
  ```

-
  ```
  Q: "5.10" as version > "5.99" as version
  A: False
  ```

The Relevance language compares the numeric values of the version segments (separated by periods), regardless of the number of digits in the segment. To be safe, always specify complete version numbers.

Inspecting the Windows Registry

You can abbreviate the root keys of the registry. For instance, instead of "HKEY_LOCAL_MACHINE\Software" you can write "HKLM\Software", etc. Here is the complete list of registry shortcuts:

HKCR	HKEY_CLASSES_ROOT
HKCU	HKEY_CURRENT_USER
HKLM	HKEY_LOCAL_MACHINE
HKU	HKEY_USERS
HKCC	HKEY_CURRENT_CONFIG

HKEY_CURRENT_USER

The BES Client runs as the LOCAL SYSTEM account and so its HKEY_CURRENT_USER branch does not match the logged in user's branch. However, it is still possible to get the logged in user's HKEY_CURRENT_USER branch of HKEY_USERS by searching through the Logon keys for the name of the current user:

```
Q: name of key whose ((it = name of current user as lowercase OR it starts with
name of current user as lowercase & "@") of (it as string as lowercase) of value
"Logon User Name" of key "Software\Microsoft\Windows\CurrentVersion\Explorer" of
it) of key "HKEY_USERS" of registry
A: S-1-5-21-1214450339-2025729265-839522115-1013
```

Be sure to include the word "key" when you want to examine a registry key. It is easy to look at an expression like this and think you are getting a valid answer:

```
Q: exists "HKEY_LOCAL_MACHINE\Software\Microsoft\Active Setup\Installed
Components\{f502aef4-a754-4c82-9f12-a5149f71ea89}" of registry
A: True
```

However, what is true here is that the string literal exists, not the key. The correct expression is:

```
Q: exists key "HKEY_LOCAL_MACHINE\Software\Microsoft\Active Setup\Installed
Components\{f502aef4-a754-4c82-9f12-a5149f71ea89}" of registry
A: False
```

Discovering Mapped Network Drives

It's easy and fast to find the names of the drives connected to the local computer:

```
Q: names of drives
A: A:
A: C:
A: D:
A: E:
A: F:
A: G:
```

But how do you find out about mapped drives?

```
Q: (selects ("ProviderName from win32_LogicalDisk")of WMI)
A: ProviderName
A: ProviderName
A: ProviderName
A: ProviderName
A: ProviderName
A: ProviderName=\\Plato\shared docs
```

Using a WMI Inspector like the one above shows that the last drive is mapped to a shared docs folder. You can correlate the drive names to the shared names as well:

```
Q: (if property "ProviderName" of it as string contains "=" then (substring
after "=" of (property "Name" of it as string) &" -- " & substring after "=" of
(property "ProviderName" of it as string)) else nothing) of select
objects("Name,ProviderName from win32_LogicalDisk")of WMI
A: G: -- \\Plato\shared docs
```

This expression finds all the mapped drives, and returns their names and their mapping.

Finding the Paging File Size

The following expression returns the paging file size in Megabytes:

```
Q: sum of (preceding texts of firsts " " of following texts of firsts ".sys " of
preceding texts of firsts "%00" of following texts of substrings "%00" of ("%00"
& value "PagingFiles" of key
"HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory
Management" of registry as string) as integer)
A: 1536
```

Discovering Services

The following will return the display name and 'imagepath' of services that should be listed under the services control:

```
Q: (display name of it, image paths of it) of services
A: Alerter, %SystemRoot%\system32\svchost.exe -k LocalService
A: Application Layer Gateway Service, %SystemRoot%\System32\alg.exe
A: Application Management, %SystemRoot%\system32\svchost.exe -k netsvcs
A: ASP.NET State Service,
%SystemRoot%\Microsoft.NET\Framework\v1.1.4322\aspnet_state.exe
A: Windows Audio, %SystemRoot%\System32\svchost.exe -k netsvcs
A: Background Intelligent Transfer Service, %SystemRoot%\system32\svchost.exe -k
netsvcs
A: Computer Browser, %SystemRoot%\system32\svchost.exe -k netsvcs…
```

See the Microsoft registry documentation reference for more information.

Determining the OS Language

To retrieve the language of the current Windows Operating System, use:

```
Q: language of version block of file "user32.dll" of system folder
A: English (United States)
```

This Relevance clause applies to all existing versions of Windows, including 95, 98, ME, XP, NT4 and 2000.

Recognizing Office Service Packs

You may need to verify an Office Service Pack before you apply an action. This is not always a trivial procedure. Information about Office XP is stored as an uninstall key in the registry, which has a name enclosed in curly brackets like this:

```
{WXYYZZZZ-6000-11D3-8CFE-0050048383C9}
```

Similarly, Office 2003 has a registry entry like this:

```
{WXYYZZZZ-6000-11D3-8CFE-0150048383C9}
```

Where:

W: Release Type = 9 (Manufacturing)

X: Edition Type = 0 or 1

YY: SKU of product

Office XP:11 (Pro), 12 (Standard), 13 (Sm Bus), 28 (Pro w/ FrontPage), ...

Office 2003: 11 (Pro), 12 (Std), 13 (Basic), CA (Sm Bus), ...

ZZZZ: Hexadecimal language identifier of product; English = 0409 (1033 decimal)

Office XP SP3

The pattern of numbers that identifies Office XP SP3 is encoded in the following Relevance expression:

-
  ```
  Q: exists key whose (value "DisplayVersion" of it as string as version >=
  "10.0.6626.0" as version AND (character 1 of it = "9" AND (character 2 of it =
  "0" OR character 2 of it = "1") AND (first 2 of following text of first 3 of it
  = "11" OR first 2 of following text of first 3 of it = "12" OR first 2 of
  following text of first 3 of it = "13" OR first 2 of following text of first 3
  of it = "28") AND (preceding text of first "}" of it ends with "6000-11D3-8CFE-
  0050048383C9")) of name of it) of key
  "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall" of
  registry
  ```

Office 2003 SP2

The pattern of numbers that identifies Office 2003 SP2 is encoded in the following Relevance expression:

-
  ```
  Q: exists key whose (value "DisplayVersion" of it as string as version >=
  "11.0.7969.0" as version AND (character 1 of it = "9" AND (character 2 of it =
  "0" OR character 2 of it = "1") AND (first 2 of following text of first 3 of it
  = "11" OR first 2 of following text of first 3 of it = "12" OR first 2 of
  following text of first 3 of it = "13" OR first 2 of following text of first 3
  of it = "CA" OR first 2 of following text of first 3 of it = "E3") AND
  (preceding text of first "}" of it ends with "6000-11D3-8CFE-0150048383C9")) of
  name of it) of key
  "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall" of
  registry
  ```

Detecting Foreign Language Service Packs

Typically, it is a simple matter to find out if a Windows Service Pack is installed by inspecting the Corrective Service Disk (CSD) version of the operating system:

-
  ```
  Q: csd version of operating system
  A: Service Pack 2
  ```

For instance, this Relevance expression will be true if Service Pack 1 or 2 is installed:

-
  ```
  Q: (csd version of it = "Service Pack 1" or csd version of it = "Service Pack
  2") of operating system
  A: True
  ```

However, both Hungarian and Polish have unconventional names for their Windows Service Packs. The following Inspectors will properly identify these anomalies:

Hungarian

-
  ```
  Q: (name of it = "WinXP" AND (it = "Service Pack 1" OR it = "Szervizcsomag 1" OR
  it = "Service Pack 2" OR it = "Szervizcsomag 2") of csd version of it) of
  operating system
  ```

-
  ```
  Q: (name of it = "Win2003" AND (it = "" OR it = "Service Pack 1" OR it =
  "Szervizcsomag 1") of csd version of it) of operating system
  ```

The first expression finds the service pack on WinXP, and the second works for Win2000 machines. They will return true if Service Pack 1 or 2 have been installed.

Polish

-
  ```
  Q: (name of it = "WinXP" AND (csd version of it = "Service Pack 1" OR csd
  version of it = "Service Pack 2" OR (csd version of it as lowercase starts with
  "dodatek" AND (csd version of it ends with " 1" OR csd version of it ends with "
  2")))) of operating system
  ```

Deconstructing XML

You can deconstruct XML files using the appropriate Inspectors. For this example, assume you have an XML file named “c:\sample.xml” that looks like this:

```
<?xml version="1.0"?>
<message>
     <to>jim@rocket_science.com
            <name>Jim Neutron</name>
            <nickname>Jimmy the Geek</nickname>
     </to>
     <to>bob@rocket_science.com
            <name>Bob Goddard</name>
            <nickname>The Bobster</nickname>
     </to>
     <from>joe@big_sky.com</from>
     <subject>Let’s do launch!</subject>
     <text>We are ready to test the new orbiter!</text>
</message>
```

Here are the results of using various XML Inspectors on this file:

-
  ```
  Q: node names of child nodes of xml document of file "c:\sample.xml"
  A: xml
  A: message
  ```

-
  ```
  Q: node names of child nodes of selects "message" of xml document of file
  "c:\sample.xml"
  A: to
  A: to
  A: from
  A: subject
  A: text
  ```

-
  ```
  Q: node names of child nodes of selects "message/to" of xml document of file
  "c:\sample.xml"
  A: #text
  A: name
  A: nickname
  A: #text
  A: name
  A: nickname
  ```

-
  ```
  Q: unique values of node names of child nodes of selects "message/to" of xml
  document of file "c:\sample.xml"
  A: #text
  A: name
  A: nickname
  ```

-
  ```
  Q: node values of child nodes 0 of selects "message/to" of xml document of file
  "c:\sample.xml"
  A: jim@rocket_science.com%0a
  A: bob@rocket_science.com%0a
  ```

-
  ```
  Q: node values of child nodes of selects "message/to/nickname" of xml document
  of file "c:\sample.xml"
  A: Jimmy the Geek
  A: The Bobster
  ```

-
  ```
  Q: node names of next siblings of selects "message/to" of xml document of file
  "c:\sample.xml"
  A: to
  A: from
  ```

-
  ```
  Q: node names of parent nodes of selects "message/to" of xml document of file
  "c:\sample.xml"
  A: message
  A: message
  ```

Using White Lists

You can use Relevance expressions to search client computers for approved applications. First, create a file named ‘whitelist.txt’ with the names of the approved applications. The names should match the registered application name as returned by the regapp Inspector. Here’s a relevance statement that outputs the names of registered applications:

-
  ```
  Q: unique values of names of regapps
  A: AcroRd32.exe
  A: Acrobat Elements.exe
  A: Ahqrun.exe
  A: CTDVDA.exe
  A: CTDVDDET.exe
  A: CTRegSvr.exe
  A: EPSONCD.exe
  A: EXCEL.EXE…
  ```

Now create a white-list file with one application name per line, like the following:

-
  ```
  Q: lines of file "c:\whitelist.txt"
  A: acrodist.exe
  A: conf.exe
  A: EXCEL.EXE
  A: IEXPLORE.EXE
  A: msconfig.exe
  A: OUTLOOK.EXE
  A: Photoshop.exe
  A: WINWORD.EXE
  A: WINZIP32.EXE
  A: wmplayer.exe
  A: wordpad.exe…
  ```

Now you can craft an expression that compares your white-listed applications with the installed applications stored in the registry. This expression outputs a list of the approved applications that exist on the client computer:

-
  ```
  Q: (lines of file "c:/whitelist.txt", unique values of names of regapps) whose
  (item 0 of it is item 1 of it)
  A: conf.exe, conf.exe
  A: EXCEL.EXE, EXCEL.EXE
  A: IEXPLORE.EXE, IEXPLORE.EXE
  A: msconfig.exe, msconfig.exe
  A: OUTLOOK.EXE, OUTLOOK.EXE
  A: Photoshop.exe, Photoshop.exe
  A: WINWORD.EXE, WINWORD.EXE
  A: WINZIP32.EXE, WINZIP32.EXE
  A: wmplayer.exe, wmplayer.exe
  A: wordpad.exe, wordpad.exe
  ```

You can test for files that are *not* approved by checking to make sure that a given registered application doesn't exist anywhere in the white-list. This is done by checking the multiplicity of non-matches. If the non-matches equal the number of lines in the white-list, then the application is nowhere on the list:

-
  ```
  Q: unique values whose (multiplicity of it = number of lines of file
  "c:/whitelist.txt") of (item 1 of it) of it whose ((item 1 of it) does not start
  with (item 0 of it)) of ((lines of file "c:/whitelist.txt"), unique values of
  names of regapps)
  A: AHQTbU.exe
  A: AcroRd32.exe
  A: Ahqrun.exe
  A: AudioCvt.exe
  A: AudioHQU.exe
  A: BrainExplorer.exe
  A: CISDS.ds
  A: CTCMSGo.exe
  A: CTDVDA.exe …
  ```

This produces a list of applications on the client computer that are not approved. This list can directly drive an Action, or it can be sent to the BES Administrator who can decide how to handle it.

Note that you could also have a black-list that could serve to identify known unapproved applications.

Appendix

Relevance Language Grammar

The grammar for the relevance language can be expressed in the Backus Naur Format as follows:

```
<primary>          :=  ( <expression> ) | string | numeral | it
<index>            :=  phrase <primary> | phrase | <primary>
<property>         :=  phrase <primary> <whose primary> <of property> |
                       primary <whose primary> <of property>
<cast>             :=  <cast> as phrase | < property >
<phrase>           :=  item | number | <expression>
<unary>            :=  exists <unary> | notExists <unary> | not <unary> | - <unary> | <cast>
<productOperator>  :=  * | / | mod | &
<product>          :=  <product> <productOperator> <unary> | <unary>
<sum>              :=  <sum> + <product> | <sum> - <product> | <product>
<relationExpr>     :=  <sum> relation <sum> | <sum>
<relation>         :=  relationOperator | relationPhrase
<relationOperator> :=  = | != | < | > | <= | >=
<relationPhrase>   :=  is | is equal to | equals | is not | is not equal to | does not equal |
                       is greater than | is not greater than | is less than | is not less than |
                       is less than or equal to | is not less than or equal to |
                       is greater than or equal to | is not greater than or equal to | contains |
                       does not contain | is contained by | is not contained by | starts with |
                       does not start with | ends with | does not end with
<andExpression>    :=  <andExpression> and <relationExpr> | <relationExpr>
<orExpression>     :=  <orExpression> or <andExpression> | <andExpression>
<tuple>            :=  <orExpression> , <tuple> | <orExpression>
<collection>       :=  <collection> ; <tuple> | <tuple>
<expression>       :=  if <expression> then <expression> else <expression> | <collection>
```

Relevance Operators

Operator	Effect	Grammatic Value
&	The string concatenation operator.	&
,	The tuple operator. Creates a tuple of objects.	,
;	The collection operator. Collects its operands into one plural result.	;
+	The sum operator.	+
-	The subtraction operator.	-
*	The multiplication operator.	*
/	The division operator.	/
=	Equivalent to the 'is' keyword.	relation
!=	Equivalent to 'is not'.	relation
<	The 'less than' operator.	relation
<=	The 'less than or equal to' operator.	relation
>	The 'greater than' operator.	relation
>=	The 'greater than or equal to' operator.	relation

Precedence and Associativity

In the Relevance language, the operator precedence is fairly standard, e.g. multiplication has a higher precedence than addition, so 3+5*2 = 3+(5*2), not (3+5)*2:

```
Q: 3+5*2
A: 13
```

Parentheses, as expected, trump the other operators:

```
Q: (3+5)*2
A: 16
```

If two operators with the same precedence act on the same object, then a choice is made to associate first with either the left or right object. Addition and subtraction are left-associative, thus, 1+2-3+4 is processed as (((1+2)-3)+4).

Casting is also left-associative, so that ‘3 as string as integer’ is interpreted as (3 as string) as integer:

```
Q: 3 as string as integer
A: 3
I: singular integer
```

The following is a list of the language elements, from highest to lowest precedence, including associativity where appropriate:

Description	Grammatic Value	Associativity
parentheses	( )	
casting operator	as	left
unary operator	exists, not exists, not, -	
products	*, /, mod, &	left
addition	+, -	left
relations	=, !=, <, <=, >, >=	
AND	and	left
OR	or	left
Tuple	,	
plural	;	left

Note there is no associativity listed for a relation, because multiple relation operators cannot appear in the same sub-expression. For example:

```
Q: 1 is 1 is 1
A: This expression could not be parsed.
```

Also, the tuple operator (comma) is right associative, but is not listed that way in the table because parentheses can change the association. For example, the first expression below is a triple, but the second is a pair:

```
Q: (1), (2), (3)
A: 1, 2, 3
```

```
Q: (1), (2,(3))
A: 1, ( 2, 3 )
```

Relevance Key Phrases

This section presents an alphabetized table of the keywords in the Relevance language, along with their grammatic values.

Keyword	Effect	Grammatic Value
a	Ignored by the relevance evaluator. Used to improve readability.	<none>
an	Ignored by the relevance evaluator. Used to improve readability.	<none>
and	The logical AND operator. Doesn’t evaluate the right hand side if the left hand side is false.	and
as	The typecast operator, used to convert one type to another.	as
contains	Returns TRUE when a string contains another string as a substring.	relation
does not contain	Equivalent to ‘not contains’.	relation
does not end with	Returns TRUE when a string does not end with the specified substring.	relation
does not equal	Equivalent to ‘is not’.	relation
does not start with	Returns TRUE when a string does not start with the specified substring.	relation
else	Denotes the alternative path in an ‘if-then-else’ statement.	else
ends with	Returns TRUE when a string ends with the specified substring.	relation
equals	Equivalent to ‘is’.	relation
exist	Returns a Boolean TRUE / FALSE indicating whether an object exists.	exists
exist no	Equivalent to ‘not exist’.	not exists
exists	Equivalent to ‘exist’.	exists
exists no	Equivalent to ‘not exist’.	not exists
if	The keyword to begin an ‘if-then-else’ expression.	if

is	Returns TRUE when two objects are equal. Note that not all objects can be tested for equality. Equivalent to the ‘=’ operator.	relation
is contained by	Returns TRUE when a string contains another string as a substring.	relation
is equal to	Equivalent to ‘is’.	relation
is greater than	The ‘>’ comparison.	relation
is greater than or equal to	The ‘>=’ comparison.	relation
is less than	The ‘<’ comparison.	relation
is less than or equal to	The ‘<=’ comparison.	relation
is not	Returns TRUE when two objects are not equal. Note that not all objects can be compared with this keyword.	relation
is not contained by	Returns TRUE when a string does not contain another string as a substring.	relation
is not equal to	Equivalent to the keyword ‘is not’ and the ‘!=’ operator.	relation
is not greater than	Equivalent to is less than or equal to or ‘<=’.	relation
is not greater than or equal to	Equivalent to is less than or ‘<’.	relation
is not less than	Equivalent to is greater than or equal to or ‘>=’.	relation
is not less than or equal to	Equivalent to is greater than or ‘>’.	relation
it	A reference to the closest direct object or ‘whose’ clause.	it
item	Used to index into a tuple. Always returns a singular value.	phrase
items	Equivalent to item, but returns a plural value.	phrase
mod	The modulo operator.	mod
not	The logical NOT operator.	relation
number	Returns the number of results in an expression.	phrase
of	Used to access a property of an object.	of

or	The logical OR operator. Doesn’t evaluate the right hand side if the left hand side is true.	or
starts with	Returns TRUE when a string begins with the specified substring.	relation
the	Ignored by the relevance evaluator. Used to improve readability.	<none>
then	Denotes the main path to take in an if-then-else expression.	then
there do not exist	Equivalent to ‘not exist’.	not exists
there does not exist	Equivalent to ‘not exist’.	not exists
there exist	Equivalent to ‘exist’.	exists
there exist no	Equivalent to ‘not exist’.	not exists
there exists	Equivalent to ‘exist’.	exists
there exists no	Equivalent to ‘not exist’.	not exists
whose	Used along with the ‘it’ keyword to filter plural results.	whose

Language History

BES 1.x

- Includes starting primitives, such as Boolean, integer, hertz, string, rope and time types.
- Includes file, application, version, folder, drive, OS, bios, registry, environment, current user Inspectors.
- Includes processor, device and RAM Inspectors.
- Includes service, network interface and IPV4 address types.
- Includes Action properties.
- Includes DMI Inspectors.
- Includes casting operators.

BES 2.x

- Adds network adapters.
- Adds Clients and Settings.
- Adds WMI.

BES 3.x

- Adds some world, string, client and service Inspectors.

BES 4.x

- Adds access control entries and lists (ACEs, ACLs).
- Extends file version properties.
- Adds floating point numbers and integer ranges.
- Adds metabase Inspectors.
- Adds network shares and port mapping.
- Adds security descriptors and identifies.
- Introduces internet firewall inspectors.

BES 5.0

- Adds connections.
- Adds Fixlet Inspectors.
- Adds media type inspectors.

BES 5.1

- If-then-else clauses gain the ability to guard against unknown vocabulary by late binding, helping to write safe cross-platform expressions.
- Introduces Introspectors that can query the language itself.
- Introduces bit sets.
- Adds file lines and folder descendents.
- Adds more firewall Inspectors.
- Adds wake-on-lan status.
- Adds XML Inspectors.

BES 6.0

- Introduces session Inspectors, including BES Actions, computers, sites, Fixlets, results, properties, users and Wizards.
- Introduces statistical Inspectors.
- Introduces multiplicities of strings and integer.
- Introduces multi-valued results (tuples).
- Adds conjunction and disjunction of Boolean lists.
- Introduces new date Inspectors, including months, years and the day of week, month and year.
- Adds regular expressions.
- Extends access control list (ACL) Inspectors.
- Improves WMI hierarchy Inspectors.
- Adds HTML-creation Inspectors.
- Adds Windows Event logs and local groups.
- Introduces application usage summaries.

Error Messages

Parsing Error

-
  ```
  Q: = )
  E: This expression could not be parsed.
  ```

This message is a catch-all for ungrammatical arrangements.

Unexpected character

-
  ```
  Q: #
  E: This expression contained a character which is not allowed.
  ```

A character that has no meaning in the language was used outside of string constants.

Strange punctuation.

-
  ```
  Q: !
  E: This expression has strange punctuation.
  ```

The expression contains a character that is a prefix of an operator, but is not an operator itself. Specifically, an exclamation point can trigger this error.

Word too long

-
  ```
  Q: key longword0123456789012345678901234567890123456789012345678901234 5 of
  registry
  E: This expression has a very long word.
  ```

A single word can only be 64 characters long, as of BES 6.0.

Phrase too long

There's also a length limit on phrases, even if the words are short.

String constant too long

```
Q:"01234567890123456789012345678901234567890123456789012345678901234567890123456
78901234567890123456789012345678901234567890123456789012345678901234567890123456
78901234567890123456789012345678901234567890123456789012345678901234567890123456
78901234567890123456789012345678901234567890123456789012345678901234567890123456
78901234567890123456789012345678901234567890123456789012345678901234567890123456
78901234567890123456789012345678901234567890123456789012345678901234567890123456
789012345678901234567890123456789012"
E: This expression has a very long string.
```

There is a current (as of BES 6.0) limit of 512 characters for the length of a string constant, after encoding any escape (%) sequences in it. This can be avoided by breaking the string up into two literals and concatenating them, like so:

```
Q:"01234567890123456789012345678901234567890123456789012345678901234567890123456
78901234567890123456789012345678901234567890123456789012345678901234567890123456
78901234567890123456789012345678901234567890123456789012345678901234567890123456
78901234567890123456789012345678901234567890" &
"1234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789012345678901234567890123456789012345678901234567890123456789
0123456789012345678901234567890123456789012345678901234567890123456789012"
```

If the string being assembled is very long, the ‘rope’ Inspector can be used to fill the gap.

Unfinished string constant

```
Q: "hello
E: A string constant had no ending quotation mark.
```

All strings need a beginning and ending pair of quotes.

String encoding error

```
Q: "%@"
E: A string constant had an improper %-sequence.
```

Percent-encoded literals are parsed to make sure they are of the form %hh, where h is a hexadecimal number. Improper codes (non-hex numbers) will trigger this error. Since the percent sign is an escape character, this means you can’t casually place it in a string literal. To get a percent sign into a string, use ‘%25’.

Integer constant too large

```
Q: 99999999999999999999
E: An integer constant was too large.
```

There is a limit on the magnitude of integer constants. As of BES 6.0, an integer constant can’t have a value greater than 2^64-1. Also, if an integer in an expression has a value between 2^63 and 2^64-1 inclusive, the error "Singular expression refers to nonexistent object" will be thrown.

Operation not defined

```
Q: 3 + "four"
E: The operator "plus" is not defined.
```

Either the operation chosen is wrong or it can't be applied to the specified types.

Singular expression required

```
Q: 1 + (2;3)
E: A singular expression is required.
```

In some contexts only singular expressions (no plurals) are allowed.

Boolean expression required

```
Q: if 5 then "boo" else "hoo"
E: A boolean expression is required.
```

In some contexts only Boolean expressions are allowed, like this:

```
Q: if true then "boo" else "hoo"
A: boo
```

Unexpected "it"

```
Q: it
E: "It" used outside of "whose" clause.
```

The "it" keyword is used in a context where it can't refer to anything.

Incompatible types

```
Q: 3; "three"
E: Incompatible types.
```

Some operations have multiple arguments that must be the same type, or at least share a base type. This example uses the plural (;) operator, which requires a list of objects with the same type. It would work, however, with a tuple operator (,):

```
Q: 3, "three"
A: 3, three
```

Tuple index out of range

```
Q: item 2 of (1,"a")
E: The tuple index 2 is out of range.
```

Tuple items are zero-based. Here the item is out of range – the legal values are 0 and 1:

```
Q: item 1 of (1,"a")
A: a
```

Tuple index not an integer literal

```
Q: item (4-3) of (1,"a")
E: This expression contained a tuple index which was not an integer literal.
```

Calculating an index into a tuple isn't allowed. You must use a literal:

```
Q: item 1 of (1,"a")
A: a
```

Singular expression refers to nonexistent object

```
Q: character of ""
E: Singular expression refers to nonexistent object.
```

Singular expressions require at least one value. To avoid this error, use a plural expression.

Singular expression refers to multiple objects

```
Q: character of "abc"
A: a
E: Singular expression refers to non-unique object.
```

Singular expressions carry an implicit restriction that there be only one value. To avoid this restriction, use a plural expression:

```
Q: characters of "abc"
A: a
A: b
A: c
```

User defined error

```
Q: if (year of current date as integer < 2006) then "Still good" else error
"Expired"
E: User-defined error: Expired
```

This error keyword returns a user-defined error.

Inspector defined error

```
Q: total duration of application usage summaries "excel.exe"
E: application usage summary inspector is disabled
```

This is an error built into the 'application usage summary' function.

Invalid operator (<&)

A sequence of legal punctuation marks turned out not to be a recognized operator.

Conversion has wrong type

When a complete expression has the wrong type for the context in which it's used, an implicit conversion is attempted: ‘as boolean’ for x-relevant-when, and ‘as string’ for most other contexts. If the conversion exists, but doesn't return the expected type, you get this error.

Value not converted

One of those implicit conversions (see above) produced no value for one of its arguments.

Cannot evaluate now

An Inspector function encountered an error it expects to be transient.

Index

A

abbreviate · 39
abbreviation · 13
ACEs · 53
ACL · 54
action · 3, 31, 32, 41
action script · 32
administrator · 3
AIX · 25
analyses · ii, 4, 30
appendfile · 32
application · 4, 23, 28, 29, 30, 31, 45, 46, 53, 54, 58
application usage · 30, 54, 58
arithmetic · ii, 21
as · 13, 47
assertions · 4, 31
associative · 48, 49
associativity · iii, 48, 49
audience · ii, 1

B

BES · 1, 2, 4, 23, 30, 31, 32, 39, 46, 53, 54, 55, 56
BES Relay · 30
binary · 21, 22
bios · 53
blacklist · 10, 16, 17
Boolean · 4, 12, 19, 22, 23, 27, 50, 53, 54, 57
branch · 23, 32, 39

C

calculations · 3
cast · 13, 47
casting · 49, 53
chart · 2, 4, 30
client · 2, 3, 4, 6, 8, 11, 16, 24, 25, 27, 30, 31, 45, 46, 53
collections · 10
colon · 10
comma · 19, 49
comparators · 11
concatenate · 13, 19, 22, 34, 35, 37, 48, 56
conditional · 23
conjunction · 10, 54
connections · 54
console · 1, 30
contains · 7, 8, 9, 10, 11, 12, 15, 16, 17, 18, 23, 29, 34, 40, 47, 50, 51, 55
current user · 27, 39, 53
customize · 4, 31, 34

D

date · 11, 13, 23, 27, 35, 36, 37, 54, 58
Debugger · 4, 5
decimal · 41
deployment · 1, 3, 4
descriptor · 28
directories · 9
disjunction · 54
divide · 3, 21
division · 21, 48
dll · 16, 17, 41
DMI · 53
DNS · 4
does not contain · 47, 50, 51
does not end with · 47, 50
does not equal · 47, 50
does not start with · 46, 47, 50
drives · 40

E

else · ii, 23, 30, 32, 40, 47, 50, 54, 57, 58
embed · 1, 31, 32
encode · 6, 42, 56
ends with · 11, 12, 24, 42, 43, 47, 50
environment · 1, 3, 27, 33, 53
equals · 47, 50
error messages · 2, 5, 23
exclamation · 55
exists · 8, 9, 10, 16, 17, 18, 20, 21, 22, 24, 27, 30, 31, 39, 42, 46, 47, 49, 50, 52, 59
exists no · 50
expressions · ii, iii, 2, 24, 34

F

file · 4, 5, 8, 9, 10, 13, 16, 17, 18, 20, 21, 27, 28, 29, 32, 40, 41, 44, 45, 46, 53, 54
file system · 27, 28, 29
filter · 7, 14, 16, 17, 52
firewall · 32, 53, 54
firmware · 25, 30
Fixlet · 1, 3, 4, 5, 31, 33, 54
floating point · 26, 38, 53
folder · 5, 8, 9, 10, 16, 17, 18, 19, 20, 21, 27, 28, 29, 32, 33, 40, 41, 53, 54
function · 36, 58, 59

G

global · 25
grammar · iii, 47, 48, 49, 50

H

hardware · 3, 11, 25, 30
hash · 4, 32
hertz · 53
hexadecimal · 41
hierarchy · 54
HKCC · 39
HKCR · 39
HKCU · 39
HKEY · 24, 32, 39, 40, 42
HKLM · 39
HKU · 39
HP · 25
http · 32
Hungarian · 43

I

identifier · 41
if-then-else · 23, 50, 52
imagepath · 41
Inspector · ii, 1, 4, 5, 6, 7, 8, 9, 19, 25, 26, 27, 28, 32, 35, 36, 37, 40, 45, 56, 58, 59
integer · 6, 13, 19, 20, 21, 23, 29, 36, 37, 40, 49, 53, 54, 56, 58
intersection · 20
Introspectors · 28, 54
is contained by · 12, 47, 51
is greater than · 11, 12, 47, 51
is greater than or equal to · 12, 47, 51
is less than · 12, 47, 51
is less than or equal to · 12, 47, 51
is not · 4, 10, 11, 12, 17, 28, 31, 32, 41, 47, 48, 49, 50, 51, 55, 57
is not equal to · 47, 51
item · 4, 19, 20, 21, 30, 46, 47, 51, 57, 58
iteration · 21

K

key · iii, 50
keyword · 7, 8, 14, 17, 19, 22, 23, 25, 36, 37, 48, 50, 51, 52, 57, 58

L

language · 1, 2, 3, 4, 5, 6, 11, 14, 19, 21, 22, 25, 26, 28, 30, 32, 38, 41, 47, 48, 49, 50, 54, 55
library · 4, 25, 26
Linux · 25
local group · 11, 54

M

Mac · 11
Macintosh · 25
mapped drives · 40
math · 6, 21
metabase · 53
microseconds · 5
minus · 22
modulo · 14, 21, 47, 49, 51
months · 21, 54
multiplication · 21, 48
multiplicity · 26, 46, 54

N

named global · 25
nodes · 44, 45
non-existent · ii, 8, 10, 16, 17, 27, 28, 56, 58
not exists · 22, 47, 49, 50, 52
now · 12, 19, 22, 27, 35, 36, 59
number · 7, 11, 17, 19, 21, 24, 25, 26, 35, 38, 46, 47, 51, 56
numbered global · 25

numeral · 47
numeric · 11, 35, 38

O

Office · iii, 27, 41, 42
operands · 48
operating system · 4, 22, 23, 30, 32, 43
operations · 57
operator · 19, 22, 27, 28, 48, 49, 50, 51, 52, 55, 57, 58
or expression · 47
OS · iii, 23, 25, 26, 41, 53
OS Language · iii, 41

P

Pack · 43
pagefile · 16, 20, 40
parentheses · 7, 15, 17, 22, 37, 49
parse · 23, 55
patch · 3
pathname · 28, 31, 33
paths · 41
pause · 31
phrase · 17, 27, 47, 51
plural · 9, 10, 20, 21, 57
Polish · 43
precedence · iii, 48
prefix · 55
Primary · ii, 6, 7
print · 13
processor · 53
productOperator · 47
profile · 32
propagation · ii, 27
property · 4, 8, 9, 11, 13, 14, 16, 27, 28, 30, 40, 47, 51
Property · ii, 11, 28, 30
property analysis · 30
punctuation · 55, 58

Q

QnA · 5, 6, 28
quotes · 56

R

RAM · 53
readability · 15, 50, 52
recursive · 31
references · 18, 27
regapp · 22, 30, 31, 45
regex · 37
registry · iii, 39
regset · 32
regular expressions · 54
relation symbols · 12
relationExpr · 47
relationOperator · 47
relationPhrase · 47
relations · ii, 11, 12
relay · 30
Relevance Debugger · ii, 5, 6, 13
remediation · 1, 6
responsiveness · 5, 21
revision · 38

S

scope · 15
security · 28, 32, 53
sequence · 56, 58
servers · 3
Service · iii, 41, 43
service pack · iii, 41, 43
Session · 4, 40
settings · 10, 53
SharedAccess · 32
shortcuts · 39
singular · 7, 8, 9, 10, 16, 19, 23, 30, 49, 51, 57
Solaris · 25
solitary · 9, 10
StandardProfile · 32
starts with · 11, 12, 16, 17, 39, 43, 47, 52
statistical · 4, 54
string · 5, 6, 7, 9, 10, 11, 13, 19, 20, 22, 23, 24, 26, 27, 28, 29, 30, 31, 32, 34, 35, 36, 39, 40, 42, 47, 48, 49, 50, 51, 52, 53, 55, 56, 59
substring · 7, 9, 10, 34, 40, 50, 51, 52
subtraction · 21, 48
subtype · 28
sum · 11, 21, 40, 47, 48
syntax · 25
SystemRoot · 41

T

then · 4, 23, 50, 52
there do not exist · 52
there does not exist · 52
there exist · 52
there exists no · 52
time · 1, 3, 4, 5, 14, 18, 19, 22, 26, 27, 30, 35, 36, 53
transient · 59
tuple · 10, 19, 20, 30, 47, 48, 49, 51, 57, 58

U

unary · 22, 47, 49
ungrammatical · 55
Unicode · 17
Unix · 11, 25
users · 3, 4, 54

V

versions · 38, 41

W

wake · 54
wake-on-lan · 54
whitelist · 5, 10, 17, 45, 46
whose · 7, 9, 10, 14, 15, 16, 17, 18, 20, 21, 24, 29, 39, 42, 46, 47, 51, 52, 57
whose-it · 15, 16, 18
WinNT · 22, 23
WinXP · 22, 23, 32, 43
wizards · 4
WMI · 40, 53, 54

X

xcopy · 16
XML · iii, 44, 54

Y

yyyymmdd · 36