

Version 8.2

*Endpoint Manager
Relevance Language Guide*



Version 8.2

*Endpoint Manager
Relevance Language Guide*



Note

Before using this information and the product it supports, read the information in “Notices” on page 65.

This edition applies to version 8, release 2, modification 0 of Tivoli Endpoint Manager and to all subsequent releases and modifications until otherwise indicated in new editions.

© **Copyright IBM Corporation 2012.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Chapter 1. Introducing the Relevance

language 1

The Scope of Relevance. 2

Using the Fixlet Debugger. 5

Using the Presentation Debugger 6

Chapter 2. Using Relevance 7

Using Relevance 7

Relevance language Overview 7

IBM Endpoint Manager Inspectors. 22

Relevance in Property Analysis. 25

Relevance in action Scripts 26

Useful Relevance Expressions 28

Appendix A. Relevance language

Grammar. 45

Relevance Operators 46

Precedence and Associativity 46

Relevance Key Phrases 47

Language History 49

BES 1.x. 49

Error Messages 51

Appendix B. Glossary 61

Appendix C. Support. 63

Notices 65

Chapter 1. Introducing the Relevance language

IBM Endpoint Manager allows large networks of computers to be easily monitored and patched in real time using **Fixlet™** technology. A Fixlet inspects a client computer and reports back to central servers. This allows a patch or an update to be efficiently applied to just those computers where it is relevant, and no others. It also enables the retrieval of various computer properties that can be collected, analyzed, charted and archived.

The heart of the Fixlet technology is the **Relevance language™** that allows authors to interrogate the hardware and software properties of your managed clients using **Inspectors**. With the Relevance language, you can write expressions describing virtually any aspect of the client environment. Some Fixlets 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 user or 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.

Dividing the labor in this fashion, using Relevance clauses to benignly inspect the client and actions to fix them (upon approval), the IBM Endpoint Manager applications provide an unprecedented blend 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 computer properties using **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.

This guide is your reference for the Relevance language. With this guide and the appropriate platform-specific *Inspector Guides*, you will be able to write Relevance expressions that can trigger Fixlet messages. You can learn how to create actions by reading the *IBM Endpoint Manager action Language Reference*.

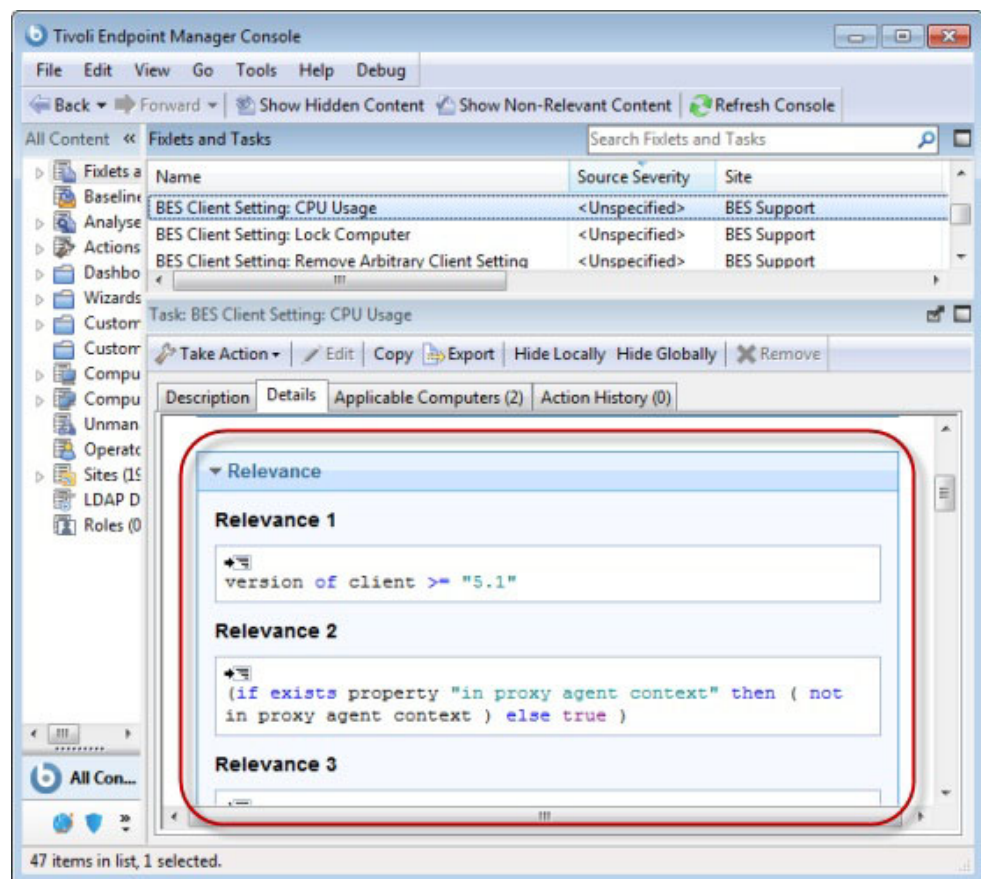
This reference is for IT managers who want to write Fixlet messages for computers managed by the IBM Endpoint Manager. The standard IBM Endpoint Manager package includes tens of thousands of pre-packaged Fixlets and associated actions, but you can expand on these offerings with custom content tuned to your specific enterprise and security configuration.

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. This will allow users to understand the Relevance clauses embedded in action scripts as well.

The Scope of Relevance

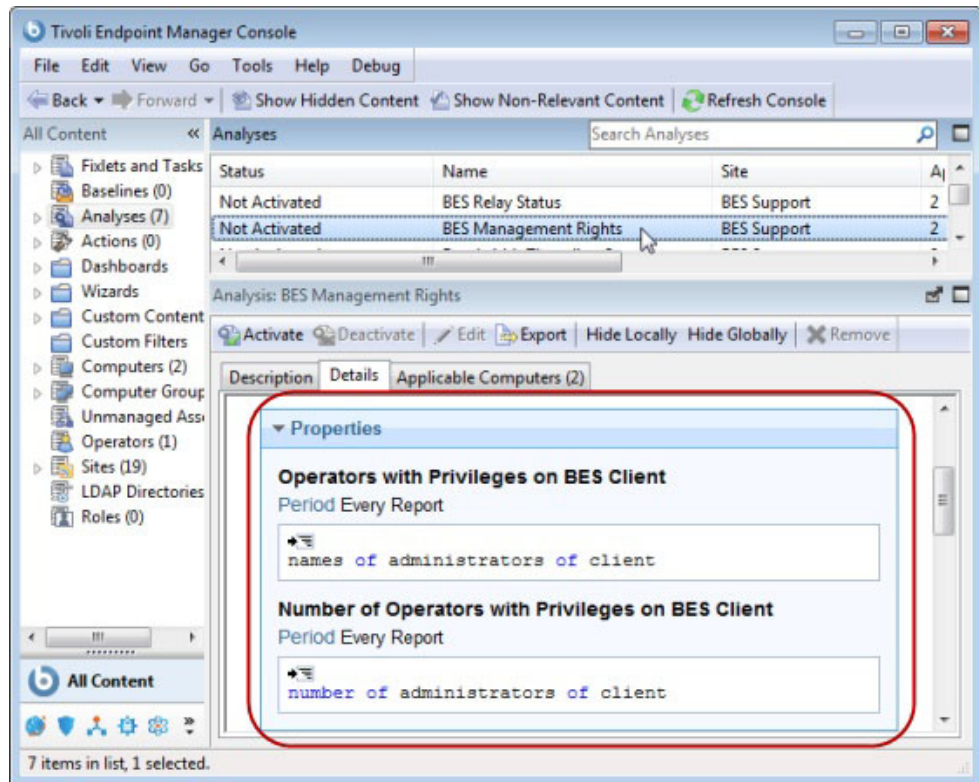
Relevance is a continuous thread used throughout all aspects of the IBM Endpoint Manager. Some of its more important manifestations include:

- **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 only when the client computer exhibits a particular state – thus the Fixlet is not displayed unless it's relevant. You can see the Relevance expression behind each Fixlet in the IBM Endpoint Manager Console: click on a **Fixlet** from the list, then look at the **Details** tab.



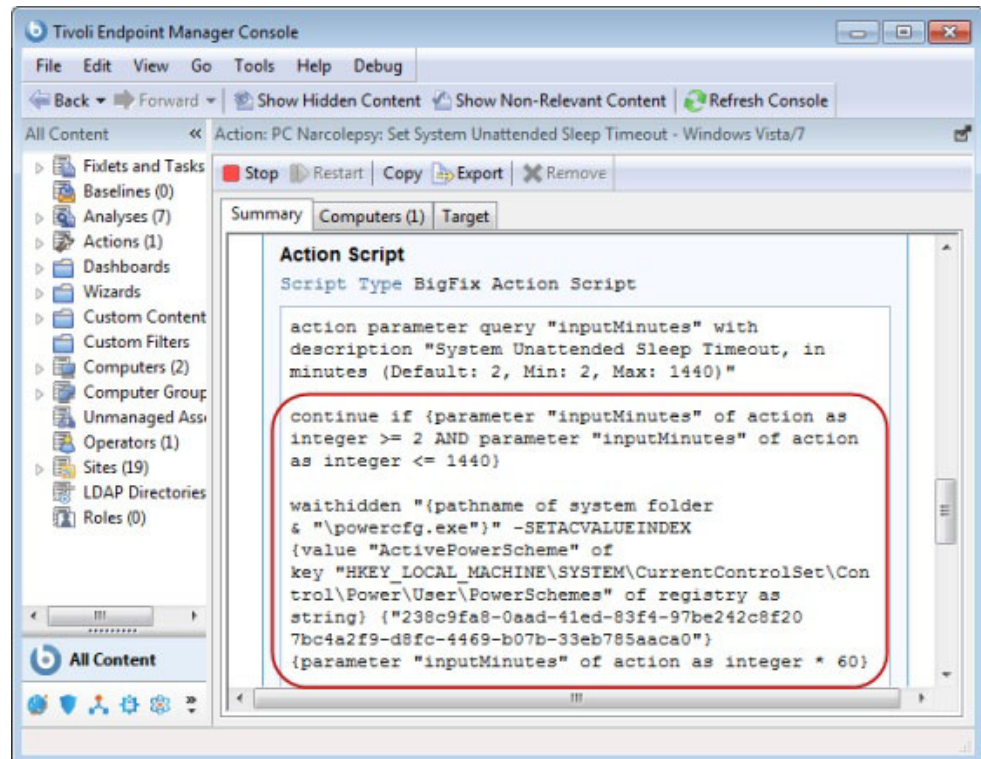
You may see more than one Relevance expression; these are all ANDed together to form the final expression.

- **Evaluating Tasks.** Tasks are functionally similar to Fixlets and use Relevance expressions to determine which computers should be targeted. Whereas Fixlets are designed with remediation in mind, Tasks are designed with continuing maintenance in mind. The main difference between Fixlets and Tasks lies in how they are judged to be "fixed". Fixlet actions will report back as fixed when they are no longer relevant, while by default Task actions will report back as fixed once all the lines of its action have completed.
- **Displaying Retrieved Properties.** From the IBM Endpoint Manager Console, you can retrieve properties of the 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 'administrators of client', 'computer manufacturer', 'brand of cpu', 'DNS Servers', 'operating system' and hundreds of others.

- **Using Relevance 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.



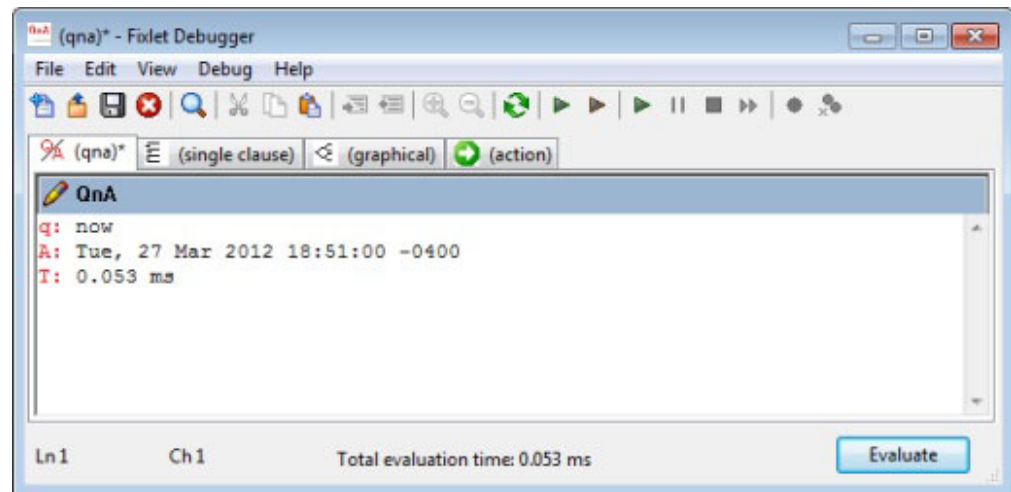
In this context, Relevance clauses are enclosed in curly brackets, such as {parameter "inputMinutes" ...} and {pathname of system folder...}. boolean Relevance clauses can be used to govern if statements and otherwise control the flow of the action script. These can be used as assertions at run-time 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 value before proceeding. Assertions make your code safer, more robust and easier to debug.

- **Reporting on IBM Endpoint Manager deployment.** Session Inspectors can be used to visualize the state of the IBM Endpoint Manager deployment itself. There are hundreds of Inspectors that can examine 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 deployment.

In the following sections, we introduce two of the primary tools for experimenting, testing and debugging your own Relevance expressions. Keep in mind that there are two distinct venues for Relevance expressions: client and session. Client Relevance allows you to inspect and repair the endpoints of your network. Session Relevance allows you to analyze your central database. These two groups don't always overlap, and viewing Relevance expressions in the wrong debugger can give incorrect results.

Using the Fixlet Debugger

For testing and debugging client relevance, there is also a stand-alone debugger called the Fixlet Debugger (previously called the Relevance Debugger or QnA) that you can use.



Here is how to use it:

1. Log in as an Administrator and run the program **FixletDebugger.exe**. It is located in **Program Files > BigFix Enterprise > BES Console > QnA**.
2. Click **Yes** to allow the program to run.
3. Type in a Relevance expression preceded by Q:, such as
Q: now
4. Click the **Evaluate** button.
5. Beneath the expression, you will see the evaluation, such as
Q: now
A: Tue, 27 Mar 2012 18:51:00 -0400
T: 0.053 ms

Note that, in addition to returning the answer, the program can also display the time (in milliseconds) it takes to process the request. This is important if you are trying to avoid time-consuming evaluations. In order to show the time, select it as an option from the **View** menu.

If an error is encountered, a message will be printed preceded by 'E:'. (Refer to the Appendix for descriptions of Relevance error messages.)

There are options in the View menu that 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 (question and answer) 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 ...

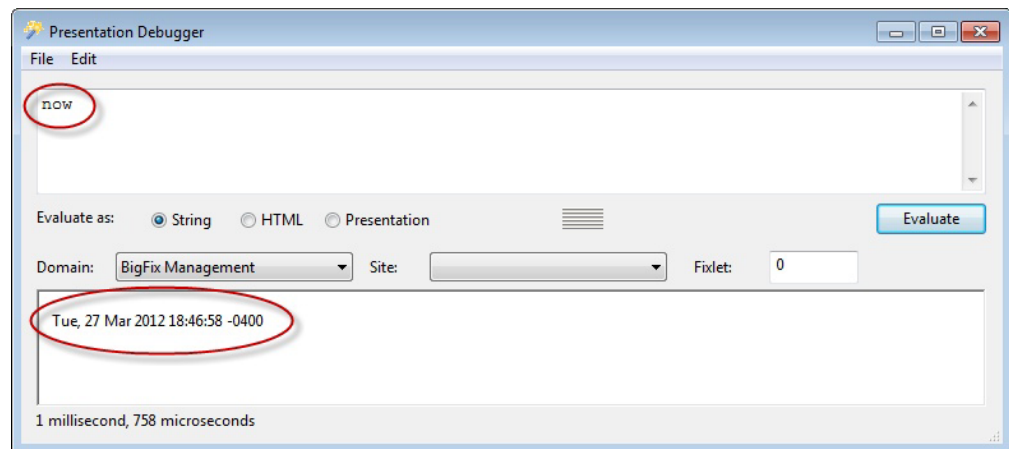
T: 1.944 ms
I: plural string
```

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

Using the Presentation Debugger

IBM Endpoint Manager includes tools to help you write and debug session relevance expressions. Here is how to install the session (also called presentation) debugger:

1. While the IBM Endpoint Manager Console is running, press Ctrl-Shift-Alt-D to bring up the **Debug** window.
2. Click the check box next to **Show Debug Menu**, at the top of the window. This installs a new menu in the Console called **Debug** that contains several handy debugging tools.
3. From the Debug menu, click on **Presentation Debugger** to open the Presentation Debugger window.



To use the debugger:

1. Type a Relevance expression in the top box. As a simple example, type the key phrase **now**. This expression extracts the current time and date from the system clock.
2. Click the **Evaluate** button.
3. In the bottom text box, the current date is displayed.

Chapter 2. Using Relevance

Using Relevance

Relevance language Overview

The Relevance language, along with the Inspector extensions, is designed to let you mine your client computers for useful information, or to see if they need remediation. Inspectors are the key phrases of the Relevance language, so let's see how they are formed. Here is a illustrative page from the Windows Inspector Guide:

Operating System

The operating system object provides access to several important properties of the system.

Creation Methods

Key Phrase	Form	Description
operating system	<i>PlainGlobal</i>	Creates the global operatingsystem object. Win, Lin, Sol, HPUX, AIX, Mac, WM, Ubu

Properties

Key Phrase	Form	Return Type	Description
<operating system> as string	<i>Cast</i>	<string>	Returns a string containing the name of the operating system concatenated with the release. Win, Lin, Sol, HPUX, AIX, Mac, WM, Ubu
boottime of <operating system>	<i>Plain</i>	<time>	Returns the time of the last restart. Win, Lin, Sol, HPUX, AIX, Mac, Ubu
build number high of <operating system>	<i>Plain</i>	<integer>	Numeric representation of the most significant 16 bits of the build number. Win, WM
build number low of <operating system>	<i>Plain</i>	<integer>	Numeric representation of the least significant 16 bits of the build number. Win, WM
csd\version of <operating system>	<i>Plain</i>	<string>	Returns the Corrective Service Disk version of the operating system. The szCSDVersion as returned by the GetVersionEx system call. The format varies depending on the installed service packs. For WinNT it contains a string such as "Service Pack 3", for Win95 it can contain a string such as "B". Win, WM
ia64 of <operating system>	<i>Plain</i>	<boolean>	Returns TRUE iff the BES Client is running on Itanium. Win:7.0, WM
mac of <operating system>	<i>Plain</i>	<boolean>	Returns TRUE if the client computer is a Macintosh. Win:8.0, Lin:8.0, Sol:8.0, HPUX:8.0, AIX:8.0, Mac:8.0, Ubu

A Inspector has one or more creation methods to define the object. Each defined object, in turn, has properties that can be inspected. Here, operating system is the name of the object and the properties include various aspects of the OS, including build numbers and boot times.

To illustrate specific Relevance elements and Inspectors, the following sections include examples using the Fixlet Debugger (using the QnA view style). If possible, run the program (FixletDebugger.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 characters, including 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 to 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.

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 (there is no a: response), 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

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 exactly one 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).

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 is much 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' refers to the list immediately to the left of the 'whose' statement (outside of a 'whose' statement, 'it' refers to the first item to the right of the parentheses).

'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, returning those initial substrings of 'banana' that contain an 'n'.

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

Q: names whose (it contains "a") of files of folder "c:"

A: at170.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 folder "z:\bar" whose(exists files of it)
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: "at170.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' always refers to a single value, and never a list.

Sets

You can convert a list returned by a plural Inspector into a mathematical set. As such, you can perform typical set operations such as union and intersection. You can create sets from individual elements, separated by semicolons:

```
intersection of (set of ("to";"be"); set of ("or";"not";"to";"be"))
```

This phrase returns the set composed of the two elements: be and to. Sets cannot be directly represented in the debugger. To see the individual items in the list, use the elements command:

```
Q: elements of intersection of (set of ("to";"be"); set of ("or";"not";
"to";"be"))
A: be
A: to
```

You can also create sets from ordinary lists. Here's an example using session inspectors that must be run in the Presentation Debugger:

```
intersection of administered computer sets of bes users whose (name of it is "joe"
or name of it is "sue")
```

The "intersection" phrase returns the set of computers administered by both Sue and Joe. Similarly, you can compute the union of sets:

```
size of union of applicable computer sets of bes fixlets whose ((source severity
of it is "Critical") and (current date - source release date of it > 7 * day)) as
floating point / size of bes computer set as floating point
```

This expression returns the ratio of computers which have at least one relevant critical Fixlet released more than 1 week ago. Note the use of the word **size** which returns the number of elements in the set.

Note: The phrases with "bes", such as "bes fixlet" and "bes computer" are **session** inspectors and will only work in the Presentation Debugger while the Console is in session. Attempting to evaluate these expressions in the Fixlet Debugger will produce an error indicating that the operator is not defined.

Sets also allow subtraction:

```
set of (1;2;3;4) – set of (1;5)
```

Returns the set composed of the elements 2,3 and 4. Note that subtracting a number not in the original set doesn't affect the result. You can convert the set back to a printable list, using the **elements** command.

```
Q: elements of (set of (1;2;3;4) – set of (1;5))
A: 2
A: 3
A: 4
```

The elements keyword iterates over the set object, returning the individual set elements as an ordinary list.

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 *nix, 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: 2012

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 Fixlet 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.

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: extrema of (1;2;3;4;5)
A: 1, 5
T: 0.127 ms
I: singular (integer, integer)

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
```

Comparing Tuples

You can directly compare tuples with the usual comparison operators, <,>,. Here are some examples:

```
Q: (1, 2, 3) < (2, 3, 4)
A: True
Q: ("a", "b", "c") < ("a", "b", "d")
A: True
```

The comparison proceeds through the elements from left to right, and the first pair of elements that fails the test ends the tuple comparison:

```
Q: ("abc", 17, 25, 4) < ("zzz", 17, 25, 4)
A: True
```

This is true even though the rest of the numbers are equal, because the test fails on the leftmost element comparison. In this sense, the comparison acts like it does with version numbers, where the leftmost digits are considered to have greater

significance. Thus, if you have control over the ordering of the tuple, you should place the most "important" items earliest in the list.

The size of the tuples must be equal for the result to be defined. For instance,

Q: (1, 2, 3) < (4, 5)

E: The operator "less than" is not defined.

Also the types of the tuple elements must match:

Q: (1, 2, 3) < (4, 5, "a")

E: The operator "less than" is not defined.

Any comparisons with <nothing> will cause an error:

Q: (nothing) < (nothing)

E: A singular expression is required.

Q: (nothing) < (1)

E: A singular expression is required.

Q: (nothing, nothing) < (1)

E: A singular expression is required.

Q: (nothing, nothing) < (1,2)

E: A singular expression is required.

Q: (nothing , 1) < (nothing , 2)

E: A singular expression is required.

Q: (nothing , 1) < (1,2)

E: A singular expression is required.

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).

Using existence with boolean logic even lets you check for things that might otherwise return an error:

Q: Exists folder "C:\doesn't exist" AND Exists files "this should normally break" of folder "c:\doesn't exist"
A: False

If-then-else

If-then-else clauses have the form:

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

If statements require both a 'then' and 'else' clause or they will throw an error.

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 IBM Endpoint Manager, 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 IBM Endpoint Manager, 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.

IBM Endpoint Manager 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 <i>number</i> of <object>
NumberedGlobal	keyword <i>number</i>
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 IBM Endpoint Manager 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 IBM Endpoint Manager Server, allowing you to view or chart the results.

Relevance in action Scripts

Viewing action Scripts

You can view an action script from the IBM Endpoint Manager Console by selecting a Fixlet or Task and clicking on the **Details** tab. For more information on the action syntax, see the *IBM Endpoint Manager 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 IBM Endpoint Manager 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 IBM Endpoint Manager 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.

Using Number Ranges

As of version 8 of TEM, you can use number ranges in a variety of ways. To generate the first four numbers, you could use an expression like the following:

```
Q: integers to (3)
A: 0
A: 1
A: 2
A: 3
```

Note that the index is zero-based, so the range goes from 0-3, not 1-3. You can use negative numbers as well:

```
Q: integers in (-2, 2)
A: -2
A: -1
A: 0
A: 1
A: 2
```

As well as listing numbers incrementally by one, you can also specify a step size:

```
Q: integers in (-10, 3, 3)
A: -10
A: -7
A: -4
A: -1
A: 2
```

This expression goes from the first value (-10) in steps of three, stopping at the first integer that doesn't exceed the final value. You can derive the final value from the length of a string, as in this example:

```
Q: integers in ((0, length of it, 2) of "00000b000100")
A: 0
A: 2
A: 4
A: 6
A: 8
A: 10
A: 12
```

Note that the length is 12, and the integers command is inclusive of the endpoints, so it includes a pointer to the end of the string.

```
Q: concatenation "," of substrings (integers in (0, length of it, 2), 2) of
  "00000b00010002000000000000000000"
A: 00,00,0b,00,01,00,02,00,00,00,00,00,00,00,00
```

Here is a more complicated example that combines two lists. The first list is numeric (1, 2, 3) and the second list is alphabetic (a, b, c). This expression combines the two to produce (1a, 2b, 3c):

```
Q: substrings (integers in (0, length of it, 2) , 2) of (concatenation of
  substrings ((item 0 of it ; item 0 of it + item 1 of it) of (positions of
  first ((length of it - 1) / 2) of it, length of it / 2), 1) of it) of
  concatenation of (("1";"2";"3"); ("a";"b";"c"))
A: 1a
A: 2b
A: 3c
```

The expression makes good use of the "item N of it" syntax for indexing into a list of objects.

Using the Bar Operator

As of version 8 of TEM, you can use the bar (|) operator to iterate through a list and find the first item that doesn't produce an error. In the following example, we assume that the file gone.txt doesn't exist, so getting its size throws an error:

```
Q: size of file "c:\gone.txt" | 10
A: 10
```

The expression skips past the file error and lands on 10.

```
Q: (size of file "c:\gone.txt" > 0) | false
A: False
```

The bar operator is looking for errors, not true or false, so it will report false if that is the first error-free phrase.

```
Q: ( (not exists file "c:\gone.txt") | (size of file "c:\gone.txt" = 0) )
A: False
```

Because the bar operator can skip through errors, it can simplify your code. Typically you check for the existence of each file before examining it:

```
(
  if
  (
    exists result (it,bes properties "Shared Groups")
    whose (exists value of it)
  )
  then
    value of result (it,bes properties "Shared Groups")
)
```

```

else
  "No Result"
)
of bes computers

```

Here is how you might write this using the bar operator:

```

(
  value of result (it, bes property "Shared Groups") | "No Result"
)
of bes computers

```

The bar operator just keeps iterating through the list until it reaches an error-free phrase, which in this case is the "No Result" message.

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

Substring index of string

As of version 8 of TEM, you can retrieve indexed substring values:

Q: substring(0, 3) of "abcdefgh"

A: abc

This expression returns the first three characters of the specified string. The starting value is zero-based, so to find a substring starting at the fourth character, use an expression like this:

Q: substring(3, 3) of "abcdefgh"

A: def

You can concatenate the substrings you find as well. For instance, to produce a series of hex bytes from a string of hex characters, use an expression like the following:

Q: concatenation ",", of substrings (integers in (0, length of it, 2), 2) of "00000b00010002000000000000000000"

A: 00,00,0b,00,01,00,02,00,00,00,00,00,00,00,00,00

This expression uses the **integers in** command, that returns a list of number pairs with a starting number (starting at 0) and a step size (2):

Q: (integers in (0, length of it, 2), 2) of "123456"

A: 0, 2

A: 2, 2

A: 4, 2

A: 6, 2

Since we want two-digit hex values here, we add the ",2" to the starting values.

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 yyyyymmdd to date:

Converting from yyyyymmdd 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.

Summing Over Time Units

As of version 8 of TEM, you can sum over different units of time to create a complex time value. This allows you to work with finer time granularity and greater accuracy. Here are some examples:

Q: sum of (second; minute; hour; day)
A: 1 day, 01:01:01
Q: sum of ((second * 5); (minute * 3))
A: 00:03:05
Q: sum of ((second * 1); (second * 2); (minute * 3); (minute * 4); (hour * 5); (hour * 6); (day * 7); (day * 8))
A: 15 days, 11:07:03
Q: sum of ((second * 5); (minute * 3); (hour * 17); (day * -23))
A: -22 days, 06:56:55
Q: sum of (second / 10)
A: 00:00:00.100
Q: sum of ((hour + hour))
A: 02:00:00

Note that summing over a time interval includes weeks, days, hours, minutes and seconds only. It will not sum months or years.

Operating on Lists of Floating Point Numbers

As of version 8 of TEM, you can sum over multiple floating point numbers in a semicolon-separated list:

Q: sum of ("1.2" as floating point; "1.5" as floating point)
A: 2.7

This is just like adding floating point numbers, but it allows you to sum up a list of multiple numbers:

Q: sum of ("1.0" as floating point; "1.1" as floating point; "1.9" as floating point)
A: 4.0

Not that, as with normal floating point addition, you need to pay attention to the digits of accuracy you use. The least accurate number in the summation determines the accuracy of the answer. If you only require one digit to the right of the decimal point, you could indicate that with a phrase like this:

Q: sum of ("1.1" as floating point; "1.06" as floating point)
A: 2.2

If you require more digits of accuracy, add zeroes.

Q: sum of ("1.10" as floating point; "1.06" as floating point)
A: 2.16

You can explicitly increase the accuracy with the **more significance** operator:

Q: sum of (more significance 1 of floating point "1.1"; "1.06" as floating point)
A: 2.16

You can also multiply a list of numbers with the product operator:

Q: product of ("1" as floating point; "2" as floating point; "3" as floating point)
A: 6

Using Wildcard Searches for Files and Folders

As of version 8 of TEM, you can use the find Inspectors to search for files and folders with wildcard searches:

Q: names of find folders "system*" of windows folder
A: system
A: system32

This expression returns all folders in the Windows folder that have "system" in their name. You can also sum over your search to find the number of relevant folders:

Q: number of names of find folders "*86*" of folder "C:\windows\winsxs"
A: 5050

This expression found 23 files with "86" in the name. Here, the asterisks are wildcard characters that stand for any set of characters (including null), and will find files named "86skidoo", "my86th" and "last86".

You can also use the "?" wildcard which matches a single character and is thus more restrictive:

Q: number of names of find folders "?86*" of folder "C:\windows\winsxs"
A: 4325

You can use multiple wild cards to expand your search:

Q: number of find folders "win*" of find folders "system*" of windows folder
A: 3

You can search for the existence of any file name at all using a naked asterisk:

Q: number of names of find files "*" of find folders "X86*" of find folders "win*" of windows folder
A: 51

This expression finds all the files in the root c: drive.


```

Q: find files "*" of folder "c:\"
A: "AUTOEXEC.BAT" "" "" "" ""
A: "boot.ini" "" "" "" ""
A: "CONFIG.SYS" "" "" "" ""
A: "IO.SYS" "" "" "" ""
A: "MSDOS.SYS" "" "" "" ""
A: "NTDETECT.COM" "" "" "" ""
A: "ntldr" "" "" "" ""
A: "pagefile.sys" "" "" "" ""
A: "test.txt" "" "" "" ""

```

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
------	---------------------

Accessing the Current User Keys:

The IBM Endpoint Manager 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

Finding Registry Keys and values:

Here's how to iterate through the names and values of keys in the registry:

Q: (names of it, it) of values of key "HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Installer\UserData\S-1-5-18\Products\FAA9C0AB723160F34BC943B9B48E7F04\Patches\30BCFA5611132E741BDE91D8B6B09CE6" of registry

A: MSI3, 1

A: State, 1

A: Uninstallable, 1

A: LUAEnabled, 0

A: PatchType, 1

A: Installed, 20110106

A: DisplayName, KB945140

A: MoreInfoURL, <http://www.microsoft.com/>

You can test for the existence of specific keys:

Q: exists value whose (name of it is "DisplayName") of key "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Installer\UserData\S-1-5-18\Products\FAA9C0AB723160F34BC943B9B48E7F04\Patches\30BCFA5611132E741BDE91D8B6B09CE6" of registry

A: True

And you can test for specific values of a key:

Q: exists value whose (name of it is "DisplayName" and it is "KB945140") of key "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Installer\UserData\S-1-5-18\Products\FAA9C0AB723160F34BC943B9B48E7F04\Patches\30BCFA5611132E741BDE91D8B6B09CE6" of registry

A: True

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: Alert, %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.

Finding the Last Write Time of Registry Keys:

You can discover the last time any given registry key was written:

```
Q: last write time of key "HKEY_LOCAL_MACHINE\SOFTWARE\BigFix\" of registry
A: Thu, 07 Jan 2010 17:32:08 -0800
```

Environment variables

Windows environment variables are available for Inspection using a phrase like this:

```
Q: value of variable "PATH" of environment
A: /usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin
```

You can find and return all the environment variables with a simple expression:

```
Q: (names of it, values of it) of variables of environment
A: BESClientactionMastheadPath, /etc/opt/BESClient/actionsite.afxm
A: BESClientConfigPath, /var/opt/BESClient/besclient.config
A: CVS_RSH, ssh
A: DISPLAY, :0
A: G_BROKEN_FILENAMES, 1
A: HISTSIZE, 1000
A: HOME, /root
A: HOSTNAME, t-rhel5-x86-1
A: INPUTRC, /etc/inputrc
A: KDEDIR, /usr
A: KDE_IS_PRELINKED, 1
A: KDE_NO_IPV6, 1
A: LANG, en_US.UTF-8
A: LD_LIBRARY_PATH, /opt/BESClient/bin
A: LESSOPEN, |/usr/bin/lesspipe.sh %25s
A: LOGNAME, root
```

As of version 8 of TEM, you can also inspect environment variables associated with processes on *nix machines:

```
Q: variables of environment of process 2186
A: BESClientactionMastheadPath = /etc/opt/BESClient/actionsite.afxm
A: BESClientConfigPath = /var/opt/BESClient/besclient.config
A: BESRelayConfigPath =
A: CONSOLE = /dev/pts/0
A: DISPLAY = :0
```

Determining Operating System Information

There are Inspectors that can interrogate the operating system for you. For instance to find the version, you might use an expression like this:

```
Q: version of operating system
A: 6.1.7600
```

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.

To find the name of the client operating system, you might use a statement like this:

```
Q: name of operating system as lowercase contains "win"
A: True
```

As of TEM version 8, there is a shortcut for this syntax.

```
Q: windows of operating system
A: True
```

Acceptable values are windows, mac, and unix.

Accessing the Task Manager (Processes)

The Task Manager can be accessed on a Windows system (as of version 8 of TEM) so that you can examine all the running processes. Here is an example that returns various properties of the specific process named SQLAGENT:

```
Q: (working set size of it / 1024, page file usage of it / 1024, user of it)
of processes whose (name of it is "SQLAGENT.EXE")
A: 6492, 9336, TESTBOX\Administrator
```

You can examine virtually every property of running processes:

```
(name of it,
(if exists image file of it then pathname of image file of it else "n/a"),
id of it,
page fault count of it,
peak working set size of it,
working set size of it,
quota peak paged pool usage of it,
quota paged pool usage of it,
quota peak nonpaged pool usage of it,
quota nonpaged pool usage of it,
page file usage of it,
peak page file usage of it,
io read count of it,
io write count of it,
io other count of it,
io read size of it,
io write size of it,
io other size of it,
creation time of it,
kernel time of it,
user time of it,
handle count of it,
session id of it,
wow64 of it,
user of it,
(if (base priority of it = normal priority) then "Normal" else
if (base priority of it = high priority) then "High" else
if (base priority of it = idle priority) then "Idle" else
if (base priority of it = realtime priority) then "Realtime" else
if (base priority of it = above normal priority) then "Above Normal" else
if (base priority of it = below normal priority) then "Below Normal" else "N/A")
)
of processes
```

When you run this expression in the Fixlet debugger, use the Single Clause tab (File > New Tab > New Single Clause Tab). You will iterate through the running processes, returning strings like this:

```
smss.exe, C:\Windows\System32\smss.exe, 284, 540, 1175552, 69632, 38360, 10648,
3568, 1624, 454656, 520192, 12, 4, 654, 53790, 4, 10990, ( Wed, 04 Apr 2012 11:06:35
-0400 ), 00:00:00.031200, 00:00:00.015600, 30, 0, False, NT AUTHORITY\SYSTEM, Normal
csrss.exe, C:\Windows\System32\csrss.exe, 408, 2778, 4558848, 1863680, 155376,
143920, 15368, 13088, 2154496, 2154496, 497, 0, 5590, 386796, 0, 262035, ( Wed,
04 Apr 2012 11:06:40 -0400 ), 00:00:01.404009, 00:00:00.218401, 646, 0, False,
NT AUTHORITY\SYSTEM, Normal
wininit.exe, C:\Windows\System32\wininit.exe, 480, 1351, 4579328, 167936, 106368,
102384, 13960, 9736, 1531904, 1847296, 1, 0, 1848, 7168, 0, 9374, ( Wed, 04 Apr
2012 11:06:45 -0400 ), 00:00:00.093600, 00:00:00.046800, 77, 0, False,
NT AUTHORITY\SYSTEM, High
```

Examining Running and Scheduled Tasks

Scheduled and running tasks on a Windows computer can be accessed and examined as of version 8 of TEM. For instance, you can find the names of currently running tasks with an expression like this:

```
Q: names of running tasks
A: SystemSoundsService
A: MsCtfMonitor
```

To see what tasks are scheduled to run and when, use an expression like the following:

```
Q: (names of it, paths of it, next run times of it) of scheduled tasks
A: AD RMS Rights Policy Template Management (Automated), \Microsoft
\Windows\Active Directory Rights Management Services Client\AD RMS Rights
Policy Template Management (Automated), ( Thu, 19 Aug 2010 03:05:32 -0700 )
A: AitAgent, \Microsoft\Windows\Application Experience\AitAgent,
( Thu, 19 Aug 2010 02:30:00 -0700 )
A: ProgramDataUpdater, \Microsoft\Windows\Application Experience
\ProgramDataUpdater, ( Thu, 19 Aug 2010 00:30:00 -0700 )
A: Consolidator, \Microsoft\Windows\Customer Experience Improvement
Program\Consolidator, ( Wed, 18 Aug 2010 22:00:00 -0700 )
A: KernelCeipTask, \Microsoft\Windows\Customer Experience
Improvement Program\KernelCeipTask, ( Thu, 19 Aug 2010 03:30:00 -0700 )
```

The following example shows how to list all of the tasks that will run at logon:

```
Q: names of scheduled tasks whose (exists definitions whose (exists
triggers whose (type of it is logon task trigger type) of it) of it)
A: AD RMS Rights Policy Template Management (Automated)
A: AD RMS Rights Policy Template Management (Manual)
A: Microsoft-Windows-DiskDiagnosticResolver
A: Logon Synchronization
A: AutoWake
```

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:

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

Similarly, Office 2003 has a registry entry like this:

```
{WXYZZZZ-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: CTDVDET.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: wmpayer.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: wmpayer.exe, wmpayer.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 IBM Endpoint Manager 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 A. 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	Grammatical 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 grammatical values.

Keyword	Effect	Grammatical 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

Keyword	Effect	Grammatical Value
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

Keyword	Effect	Grammatical Value
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.

TEM 7.0

- Adds new session Inspectors.
- Adds distinguished name Inspectors.
- Adds firewall Inspectors.
- Adds integer set Inspectors.
- Adds IPv6 address and conversion Inspectors

TEM 8.0

- Introduces Scheduled Task and Task Manager Inspectors.
- Adds wildcard searches for files and folders.
- Adds indexed substring Inspectors.
- Adds range operators.
- Introduces sum of time Inspectors.
- Introduces sum and product of operators on floating point lists.
- Introduces new OS operators and environment variable Inspectors.

Error Messages

"It" used outside of "whose" clause

This is a potentially confusing error message because it is perfectly legal to use **it** without a **whose** clause as long as you form the syntax correctly. This message just means that interpreter does not know what **it** refers to, typically due to a syntax error. For example:

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

Here the "it" keyword is used in a context where it can't refer to anything. Here is another example:

```
Q: system folder (name of it & pathname of it)
E: "It" used outside of "whose" clause.
```

The error indicates that the interpreter does not know what **it** is, because the statement is formulated incorrectly. The correct statement would be:

```
(name of it & pathname of it) of system folder
```

To ensure that **it** points toward an object, you must make sure that you include **of <object>** after the statement.

A boolean expression is required

This error message is produced when a statement requires a boolean value to evaluate, but instead the receives a different type. The Relevance language is

strongly typed, and inattention to proper type matching is the source of most bugs. Among other things, a boolean type is required after **if** and **whose** statements. For example:

Q: Names of files whose (version of it = "5") of system folder

This example works because the value in the parenthesis is a boolean. Otherwise you get this error, as in this incorrect example:

Q: Names of files whose (version of it) of system folder

E: A boolean expression is required.

Here, the parenthetical statement doesn't evaluate to a boolean, so the expression fails with an error.

The test for an **if-then** statement must also be boolean. You might think a non-zero number would evaluate to true, but types must be carefully matched, and in the Relevance language, a number is a number, and not a boolean:

Q: if 1 then "boo" else "hoo"

E: A boolean expression is required.

One (1) is a number; use a boolean instead:

Q: if 1=1 then "boo" else "hoo"

A: boo

Which is the same as

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

A: boo

Along the same lines, you might think to test for an executable file like this:

Q: If regapp "besclient.exe" then version of regapp "besclient.exe" as string
else "N/A"

E: A boolean expression is required.

Instead, you should check for the *existence* of the file, which is boolean:

If exists regapp "besclient.exe" then version of regapp "besclient.exe" as string
else "N/A"

A singular expression is required

This error message can result from trying to compare two lists or a list to an object. In general, all comparisons must be made between two singular objects. For instance:

Q: Versions of files of folder "c:\temp2" = versions of files of folder "c:\temp"
e: A singular expression is required.

Returns the error because comparing two lists is undefined. This will result in an error even if both folders contain exactly the same files. Similarly,

Versions of files of folder "c:\temp" = "4.5"

Gives the same error because you can't compare a list to a single value. You will get this error even if there is only one file in the folder "c:\temp" whose version is 4.5. If you want to iterate through multiple comparisons, use the **whose** statement:

versions of files whose (version of it = "4.5") of folder "c:\temp"

You can also get this error when you try to operate on two objects with different plurality, as in this example where we attempt to add a number and a list:

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

A string constant had an improper %-sequence

You can insert characters into a string by entering a percent sign and then the ASCII hex value of the character. 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.

When you use a percent sign in a string, the relevance engine looks at the next two characters to see if they correspond to an ASCII hex value. For instance:

Q: "html uses %3Cangle brackets%3E"
A: html uses <angle brackets>
Q: "This is the %22truth%22"
A: This is the "truth"

However, a string with a percent sign in it will return this error if either of the following conditions are true:

- There are less than two characters in the string after the percent sign.
- The two characters following the percent sign aren't valid hexadecimal characters (0-9, a-f, A-F).

You won't get the error message if the percent sign is followed by letters or numbers, even if they don't correspond to a valid ASCII value (00-7F). For instance:

Q: "html uses %3Cangle brackets%9E"
A: html uses <angle brackets%9e

This may not give the desired result, because 9E is not a valid ASCII character, but it doesn't result in an error. However, this does:

Q: "html uses %3Cangle brackets%9"
E: A string constant had an improper %-sequence.

This returns the error because the percent is followed by only one character.

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':
exists "c:\testfolder\percent%25.txt"

This expression returns true if you have a file named percent%.txt in the given folder. Note that this string will print out differently in the Fixlet debugger context:

q: "c:\testfolder\percent%25.txt"
A: c:\testfolder\percent%25.txt

This is because the value is re-inserted into the text until the final invocation, to ensure that a naked percent sign never gets exposed.

A string constant had no ending quotation mark

All strings need a beginning and ending pair of quotes. This error message indicates that there is an odd number of quotation marks in the expression. For instance:

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

Invalid operator (<&)

A sequence of legal punctuation marks does not resolve to be a recognized operator.

No inspector context

Certain inspectors can only be evaluated by the client and therefore will not work in the Fixlet debugger. If you try to evaluate one of these in the Fixlet debugger, you will receive **No Inspector Context**. A common example is something like:

Q: pending restart
E: No inspector context.

In general, in order to evaluate statements that return this error you must define them as retrieved properties from the IBM Endpoint Manager Console.

Phrase too long

There is a 64-character length limit on property names.

Singular expression refers to non-unique object

This error message arises when you try to query a singular property of multiple objects. Relevance expressions require a match between the number of items in the argument and the plurality of the Inspector. For instance:

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

This expression returns a single character, as requested by the singular Inspector **character**. However, the extra characters in the argument then generate the error. To avoid this error, use a plural expression:

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

Every Inspector has both a singular and plural version. They are both listed for each Inspector in the appropriate guides.

To return a single value, you must query a single object.

Singular expression refers to nonexistent object

This is one of the most common error messages generated. It usually results from querying a property of an object that does not exist, or querying a non-existent property of an object. For instance:

Q: version of file "mslib.dll" of folder "c:\temp"
E: 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, such as **characters**. It won't return a value, since none is available, but it won't generate an error, either.

For this particular expression, the error is generated if any of the following are true:

- The folder "c:\temp" does not exist.
- There is no file named "misspelled.dll" in the folder "c:\temp".
- The file "misspelled.dll" located in the folder "c:\temp" does not have a version.

You could craft an expression that will not return an error by using plurals:

Q: versions of files "mslib.dll" of folders "c:\temp"

However, the better idea is to check for existence first:

Q: if (version of file "mslib.dll" of folder "c:\temp") then (version of file "mslib.dll" of folder "c:\temp" as string) else "n/a"

The operator "<operator>" is not defined

This error occurs when the operation chosen is wrong or it can't be applied to the specified types.

Q: 3 + "four"

E: The operator "plus" is not defined.

You also receive this message if you use a word that the Relevance interpreter does not recognize. Here are some examples:

Q: Exists executable "file_name.exe" of system folder

E: The operator "executable" is not defined.

Here the word 'executable' is not a valid command in the relevance language.

The error is also generated when a non-existent property is queried:

Q: Version of key "HLKM/Software" of registry

E: The operator "version" is not defined.

Here, even though the relevance language knows the word 'version', it does not recognize it as a valid property of **registry key**, and therefore generates this error.

The Operator "String" is not defined

This error occurs on the client machine (not in the debugger) when there is no way to cast the result as a string, so it can't be represented. For instance:

set of (1;2;3)

This expression refers to a legitimate set, but the Relevance language doesn't know how to represent it in the debugger. To see the set, use a syntax like this:

Q: elements of set of (1;2;3)

A: 1

A: 2

A: 3

Note, however, that although the client gets a string error, the Fixlet debugger will produce a different error:

Q: set of (1;2;3)

E: This expression evaluates to an unrepresentable object of type "integer set"

The tuple index <index> is out of range

Tuples can be accessed by an index, but it must not exceed the range of the data:

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

This expression contained a character which is not allowed

This error message is given when the relevance interpreter finds a character that it does not recognize. For instance:

Q: {pathname of regapp "besclient.exe"}
E: This expression contained a character which is not allowed.

Curly braces are valid tokens in the action language, but not in relevance. For use in the debugger, you need to drop the braces:

Q: pathname of regapp "besclient.exe"
A: C:\Program Files (x86)\BigFix Enterprise\BES Client\BESClient.exe

Similarly:

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

Here, a character that has no meaning in the language was used outside of string constants.

This expression contained a tuple index which was 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

This expression could not be parsed

The first step of interpreting a relevance statement is parsing the expression into its various

components. This error results from a failure of the parsing engine. This is typically

caused by unmatched parentheses or by syntax errors involving certain reserved words used by

the parsing engine. Reserved words are syntactical statements like **of**, **and**, **equals**, and so

on.

Here are a couple of examples:

Q: =)
E: This expression could not be parsed.
Q: Name of (file whose (version of it = "2.6") of system folder
E: This expression could not be parsed.

Exists file "name" of or system folder

This expression evaluates to an unrepresentable object of type "`<type>`"

```
Q: Key "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\DirectX" of registry
E: This expression evaluates to an unrepresentable object of type "registry key"
```

```
Q: exists Key "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\DirectX" of registry
A: True
```

```
Q: value "Version" of key "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\DirectX" of registry
A: 4.09.00.0904
```

[illegible][illegible][illegible]

This expression has a very long word

Q: key longword01234567890123456789012345678901234567890123456789012345 of registry
E: This expression has a very long word.

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

This expression has 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. A lone exclamation point can trigger this error.

User-defined error

You may want to define your own set of error messages. You can do this with the error command:

Q: error "oops"
E: User-defined error: oops

A more useful formulation might be:

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

The **error** keyword allows you to set your own, user-defined error codes and implement them in the **else** clause of an **if/then/else** statement.

Value not converted

The Relevance language attempts to implicitly convert results to a string, number or boolean value. This error indicates that the value could not be converted.

See "Conversion has wrong type" and "The operator 'string' is not defined", elsewhere in this list.

Appendix B. Glossary

action Password—See signing password.

action Scripting Language—The language used for crafting action scripts. Action can be crafted in different scripting languages, including AppleScript and Unix shells.

BigFix Enterprise Suite (BES)—The previous name for IBM Endpoint Manager.

Client—Software installed on each networked computer to be managed under the IBM Endpoint Manager. The Client accesses a pool of Fixlets, checks the computer it is installed on for vulnerabilities, and sends the Server a message when such a condition occurs. Previously known as the BES Client, it is now known as the IBM Endpoint Manager Client, or simply Client.

Console—A management program that provides an overview of the status of all the computers with the Client installed in the network, identifying which might be vulnerable and offering corrective actions. Previously known as the BES Console, it is now known as the IBM Endpoint Manager Console, or simply Console.

Custom Site—You can create your own custom content and host it in a custom site. This can only be done by a Master Operator that has been granted the rights to create custom content (use the Admin program to allocate these users).

Fixlet—A mechanism for targeting and describing a problematic situation on a computer and providing an automatic fix for it.

Fixlet servers—Web servers offering Fixlet site subscriptions. They can be either internal to the enterprise network or external to the network (if direct external web access is allowed).

Fixlet site—A trusted source from which the Client obtains Fixlets.

IBM Endpoint Manager database—A component of the system that stores data about individual computers and Fixlets. The IBM Endpoint Manager Server's interactions primarily affect this database, which runs on SQL Server.

IBM Endpoint Manager—A preventive maintenance tool for enterprise environments that monitors computers across networks to find and correct vulnerabilities with a few simple mouse-clicks.

Management Rights—Ordinary Console Operators can be limited to a specified group of computers. These limits represent the management rights for that user. Only a Site Administrator or a Master Operator can assign management rights.

Master Operator—A Console Operator with administrative rights. A Master Operator can do almost everything a Site Administrator can do, with the exception of creating new operators.

masthead—Files containing the parameters of the IBM Endpoint Manager process, including URLs that point to where trusted Fixlet content is available. The IBM Endpoint Manager Client brings content into the enterprise based on subscribed mastheads.

Operator—A person who operates the IBM Endpoint Manager Console. Ordinary operators can deploy Fixlet actions and edit certain computer settings. Master Operators have extra privileges, among them the ability to assign management rights to other operators.

Relay—This is a Client that is running special server software. Relays spare your server and the network by minimizing direct server-client downloads and by compressing upstream data. Relays are automatically discovered by Clients, which dynamically choose the best Relay to connect to. Previously known as the BES Relay, it is now known as the IBM Endpoint Manager Relay, or simply Relay.

Relevance Language—The language in which relevance expressions are written. Relevance expressions query the client computer to determine if remediation is desired. Remediation may consist of an action script which can also use Relevance expressions, ensuring that the item being fixed is the same as the item that was initially identified.

Server—A collection of interacting applications (web server, CGI-BIN, and database server) that coordinates the relay of information to and from individual computers in the IBM Endpoint Manager system. The server processes may be hosted by a single server computer or segmented to run on separate server computers or replicated on redundant servers. Previously known as the BES Server, it is now known as the IBM Endpoint Manager Server, or simply Server.

Signing password—The password (specified when the IBM Endpoint Manager system was installed) used by a Console operator to sign an action for deployment. It is called the *action* password in the Console interface.

Site Administrator —The only IBM Endpoint Manager Console Operator with the right to create new Operators.

Appendix C. Support

For more information about this product, see the following resources:

- Tivoli Endpoint Manager Support site
- Tivoli Endpoint Manager wiki
- Knowledge Base
- Forums and Communities

Notices

This information was developed for products and services offered in the U.S.A.

IBM® may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web

sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
2Z4A/101
11400 Burnet Road
Austin, TX 78758 U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the "Web at Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Adobe, Acrobat, PostScript and all Adobe-based trademarks are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

ITIL is a registered trademark, and a registered community trademark of the Office of Government Commerce, and is registered in the U.S. Patent and Trademark Office.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java[™] and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

Linear Tape-Open, LTO, the LTO Logo, Ultrium, and the Ultrium logo are trademarks of HP, IBM Corp. and Quantum in the U.S. and other countries.



Printed in USA