



Using Expressions

Network Registrar provides enhanced client-class support. You can now place a request into a client-class based on the contents of the request, without having to register the client in the client database. Also, you can now place requests in a client-class based on the number of a subscriber's active leases, allowing limitations on the level of service offered to various subscribers. This is possible through the special DHCP options processing using expressions.

You can set the limitation on subscriber addresses based on values in the DHCP *relay-agent-info* option (option 82, as described in RFC 3046). These values do not need to reveal any sensitive addresses. You can create values that relate an individual to a subscriber by creating an expression that evaluates the incoming DHCPDISCOVER request packets against option 82 suboptions (*remote-id* or *circuit-id*) or other DHCP options. The expression is a series of *if* statements that return different values depending on what is evaluated in the packet. This, in effect, calculates the client-class in which the subscriber belongs, and limits address assignment to the scope of that client-class.



Note

Expressions are not the same as DHCP extensions. Expressions are commonly used to create client identities or look up clients. Extensions (see [Chapter 28, “Using Extension Points”](#)) are used to modify request or response packets. The expressions described here are also not the same as regex.

Using Expressions

Expression processing is used in several places:

- Calculating a client-class—*client-class-lookup-id*. This expression determines the client-class based on the contents of the incoming packet.
- Creating the key to look up in the client-entry database—*client-lookup-id*. This accesses the client-entry database with the key resulting from the expression evaluation.
- Creating the ID to use to limit clients of the same subscriber—*limitation-id*. This is the ID to use to check if any other clients are associated with this subscriber.

This kind of processing results in this scenario:

1. The DHCP server tries to get a client-class based on a *client-class-lookup-id* expression. If it cannot calculate the client-class, it uses the usual MAC address method to look up the client.
2. If the server can calculate the client-class, it determines if it needs to do a client-entry lookup, based on evaluating a *client-lookup-id* expression that returns a *client-lookup-id*. If it has such an ID, it uses it to look up the client. If it does not have such an ID, it uses the calculated client-class value to assign addresses.

3. If the server uses the *client-lookup-id* and finds a client-entry, it uses the data for the client. If it cannot find a client-entry, it uses the calculated or default client-class data.

You set the upper limit on assigned addresses to clients on a network or LAN segment having an identical *limitation-id* value on the policy level. Set this upper limit as a positive integer using the *limitation-count* attribute for the policy.

The values to set for limiting IP addresses to subscribers are:

- For a policy, set the *limitation-count* attribute to a positive integer.
- For a client-class, set the *limitation-id* and *client-lookup-id* attributes to an expression, and set the *over-limit-client-class-name* attribute to a client-class.
- For a client, set the *over-limit-client-class-name* attribute to a client-class.

The expressions to use are described in the “[Creating Expressions](#)” section.

Entering Expressions

You can include simple expressions as such in the attribute definition, or include more complex ones in an expression file and reference the file in the attribute definition. Either way, the maximum allowable characters is 16 K. Simple expressions must adhere to these rules when you enter them in the CLI:

- They must be limited to a single command line.
- The entire expression must be enclosed in double quotes (" ").
- Embedded double quotes must be escaped with a backslash (\).

Here is an example of a simple expression to set the *client-class-lookup-id*:

```
"\"limit\""
```

Here is a slightly more extensive example to set the client-class *limitation-id*:

```
"(request option 82 \"circuit-id\")"
```

You must enter any more complex expressions, that are not limited to one line or that you want to format for comprehension, in a file and reference it in the attribute definition prefixed by the “at” symbol (@):

```
@cclookup.txt
```

The syntax of the expression in the file does not have the extra requirements (as to spacing and escaping of characters) of the simple expression. It can also include comment lines, prefixed by the pound sign (#), double-slash (//), or a semicolon (;), and terminated at the end of line. For example, in the *cclookup.txt* file:

```
// Expression to calculate client-class based on remote-id
(try (if (equal (request option "relay-agent-info" "remote-id") (request chaddr))
         "cm-client-class"
         "cpe-client-class")
     "<none>")
```

Creating Expressions

Using DHCP expressions, you can retrieve, process, and make decisions based on data in incoming DHCP packets. You can use them for determining the client-class of an incoming packet, and create the equivalence key for option 82 limitation support. They provide a way to get information out of a packet and individual options, a variety of conditional functions to allow decisions based on information in the packet, and data synthesis capabilities where you can create a client-class name or key.

The expression to include in an expression file that would describe the example in the “[Typical Limitation Scenario](#)” section on page 23-13 would be:

```
// This begins the try function
(try
  (or (if (equal (request option "relay-agent-info" "remote-id") (request chaddr))
    "cm-client-class")
    (if (equal (substring (option "dhcp-class-identifier") 0 6) "docsis")
      "docsis-cm-client-class")
    (if (equal (request option "user-class") "alternative-class")
      "alternative-cm-client-class")
  )
  <none>
)
// This ends the try function
```

This uses the **or** function and evaluates three **if** functions. In a simpler form, you can calculate a client-class and include this expression in the ccllookup.txt file.

```
// Expression to calculate client-class based on remote-id
(try (if (equal (request option "relay-agent-info" "remote-id") (request chaddr))
  "cm-client-class"
  "cpe-client-class")
  "<none>")
```

Refer to this file to use the expression to set the client-class lookup ID for the server:

```
nrcmd> dhcp set client-class-lookup-id=@ccllookup.txt
```

You can generate a limitation key by trying to get the *remote-id* suboption from option 82, and if unable, to use a standard MAC blob key. Include an expression in a file and set the limitation ID to it in the cclimit.txt file:

```
// Expression to use remote-id or standard MAC
(try (request option "relay-agent-info" "remote-id") 00:d0:ba:d3:bd:3b)

nrcmd> client-class name limitation-id=@cclimit.txt
```

Expression Syntax

Expressions consist solely of functions and literals. Its syntax is similar to Lisp's. It follows many of the same rules and uses Lisp function names where possible. The basic syntax is:

```
(function argument-0 ... argument-n)
```

A more useful example is:

```
(try (if (equal (request option "relay-agent-info" "remote-id") (request chaddr))
  "cm-client-class"
  "cpe-client-class")
  "<none>")
```

This example compares the *remote-id* suboption of the *relay-agent-info* option (option 82) with the MAC address in the packet, and if they are the same, returns “cm-client-class,” and if they are different, returns “cpe-client-class.” (If the expression cannot evaluate the data, the **try** function returns a “<none>” value—see the “[Expressions Can Fail](#)” section on page 24-5.) The intent is to determine if the device is a cable modem (where, presumably, the *remote-id* equals the MAC address) and, if so, put it into a separate client-class than a customer’s premises equipment, or PC. Note that both functions and literals are expressions. The previous example shows a function as an expression. For literals, see the “[Literals in Expressions](#)” section on page 24-4.

Expression Datatypes

The datatypes that expressions support are:

- Blob—Counted series of bytes, with a minimum supported length of 1 KB.
- String—Counted series of NVT ASCII characters, not terminated by a zero byte, with a minimum supported length of 1 KB.
- Signed integer—32-bit signed integer.
- Unsigned integer—32-bit unsigned integer.

Note that there is no IP address datatype; an IP address is a 4-byte blob. All numbers are in network byte order. See the “[Datatype Conversions](#)” section on page 24-16.

Literals in Expressions

A variety of literals are included in the expression capability:

- Signed integers—Normal numbers that must fit in 32 bits.
- Unsigned integers—Normal unsigned numbers that must fit in 32 bits.
- Blobs—Hex bytes separated by colons. For example, 01:02:03:04:05:06 is a 6-byte blob with the bytes 1 through 6 in it. This is distinct from “01:02:03:04:05:06” (a 17-byte string). The string is related to the blob by being the text representation of the blob. For example, the expression (**to-blob "01:02:03"**) returns the blob 01:02:03. Note that you cannot create a literal representation of a one-byte blob, as 01 will turn into an integer. To get a one-byte blob containing a 1, you would use the expression (**substring (to-blob 1) 3 1**). The 3 indicates the offset to extract the fourth byte of the 4-byte integer (00:00:00:01), with the 1 being the number of bytes extracted, with a result of “01.”
- String—Characters enclosed in double quotes. For example, “example.com” is a string, as is “01:02:03:04:05:06.” To place a quote in a literal string, escape it with a backslash (\), for example:


```
"this has one \"quote"
```

Integer literals (signed and unsigned) are assumed to be in base10. If they start with a 0, they are considered octal; if they start with 0x, they are considered hexadecimal. Some examples of literals:

- “hello world” is a string literal (and a perfectly valid expression).
- 1 is an unsigned integer literal (also a perfectly valid expression). It contains 4 bytes, the first three of which are zero, and the last of which contains a 1 in the least significant bit.
- 01:02:03 is a blob literal containing three bytes, 01, 02, and 03.
- -10 is a signed integer literal containing four bytes with the twos-complement representation of decimal -10.

Expressions Return Typed Values

With few exceptions, the point of an expression is to return a value. The expression configured to determine a client-class is configured in the DHCP server property *client-class-lookup-id*. When this expression is evaluated, the DHCP server expects it to return a string containing the name of a client-class, or the string `<none>`.

Every function returns a value. The datatype of the value may depend on the datatype of the argument or arguments. Some expressions only accept arguments of a certain datatype; for example:

```
(+ argument0 argument1)
```

In most cases, a function that requires a certain datatype for a particular argument tries to convert the argument that it gets to the proper datatype. For example, `(+ "1" 2)` returns 3, because it successfully converts the string literal "1" into a numeric 1. However, `(+ "one" 2)` causes an error, because "one" does not convert successfully into a number. In general, the expression evaluator tries to do the right thing as much as possible when making datatype conversion decisions.

Expressions Can Fail

While some of the functions that make up an expression operate correctly on any datatype or value, many do not. In the previous section, the `+` function would not convert the string literal "one" into a valid number, so the evaluation of that function failed. When a function fails to evaluate, its calling function also fails, and so on, until the entire expression fails. A failed expression evaluation has different consequences depending on the expression involved. In some cases, it can cause the packet to be dropped, while in others it only generates a warning message.

You can prevent the evaluation from failing by using the `(try expression failure-expression)` function. The `try` function evaluates the expression and, if successful, the value of the function is the value of the *expression*. If the evaluation fails (for whatever reason), the value of the function is the value of the *failure-expression*. The only situation where a `try` function itself fails is if the *failure-expression* evaluation fails. Thus, you should be careful what expression you define as a *failure-expression*. A string literal is a safe bet. Thus, protecting the evaluation of the *client-class-lookup-id* with a `try` function is a good idea. The previously cited example shows how this can work:

```
(try (if(equal (request option "relay-agent-info" "remote-id") (request chaddr))
        "cm-client-class"
        "cpe-client-class")
     "<none>")
```

If evaluating the `if` function fails in this case, the value of the *client-class-lookup-id* expression is `<none>`. It could have been a client-class name instead, of course.

Expression Functions

[Table 24-1](#) lists the expression functions. Expressions must be enclosed in parentheses.

Table 24-1 Expression Functions

Function	Example
Description	
(and <i>arg1</i> ... <i>argn</i>)	(and "hello" "world") returns "world" (and (request option 82 1) (request option 82 2)) returns option-82 sub-option 2 if both option-82 sub-option 1 and sub-option 2 are present in the request
	The and function returns a value that is the datatype of <i>argn</i> or null. It evaluates its arguments in order from left to right (the arguments can evaluate to a datatype). If any argument evaluates to null, it stops evaluating the arguments and returns null. Otherwise, it returns the value of the last argument, <i>argn</i> .
(as-blob <i>expr</i>)	(as-blob "hello world") returns the blob 68:65:6c:6c:6f:20:77:6f:72:6c:64
	The as-blob function treats <i>expr</i> as if it were a blob. If <i>expr</i> evaluates to a string, the bytes that make up the string become the bytes of the blob that is returned. If <i>expr</i> evaluates to a blob, that blob is returned unmodified. If <i>expr</i> evaluates to either kind of integer, a 4-byte blob containing the bytes of the integer is returned. (See Table 24-2 on page 24-16 .)
(as-sint <i>expr</i>)	(as-sint ff:ff:ff:ff) returns -1 (as-sint 2147483648) returns an error
	The as-sint function treats <i>expr</i> as if it were a signed integer. If <i>expr</i> evaluates to a string or blob of 4 bytes or less, the function returns a signed integer constructed out of those bytes (if longer than 4 bytes, it returns an error). If <i>expr</i> evaluates to a signed integer, it returns the value unchanged; if an unsigned integer, it returns a signed integer with the same bit value. (See Table 24-2 on page 24-16 .)
(as-string <i>expr</i>)	(as-string 97) returns "a" (as-string 68:65:6c:6c:6f:20:77:6f:72:6c:64) returns "hello world" (as-string 0) returns an error.
	The as-string function treats <i>expr</i> as if it were a string. If <i>expr</i> evaluates to a string, it returns that string. If <i>expr</i> evaluates to a blob, it returns a string constructed from the bytes in the blob, unless they are nonprintable ASCII values, which returns an error. If <i>expr</i> evaluates to an integer, it considers its value to be the ASCII value for a single character and returns a string consisting of that one character, unless it is nonprintable, which returns an error. (See Table 24-2 on page 24-16 .)
(as-uint <i>expr</i>)	(as-uint -2147483648) returns the unsigned integer 2147483648 (as-uint -1) returns the unsigned integer 4294967295 (as-uint ff:ff:ff:ff) returns the unsigned integer 4294967295
	The as-uint function treats <i>expr</i> as if it were an integer. If <i>expr</i> evaluates to a string or blob of 4 bytes or less, it returns an unsigned integer constructed from those bytes; if longer than 4 bytes, it returns an error. If the result is an unsigned integer, it returns the argument unchanged; if a signed integer, it returns an unsigned integer with the same bit value (see Table 24-2 on page 24-16).
(ash <i>expr</i> <i>shift</i>)	(ash 00:01:00 1) returns the blob 00:02:00 (ash 00:01:00 -1) returns the blob 00:00:80 (ash 1) returns the unsigned integer 2
	The ash function returns an integer or blob with the bits shifted by the <i>shift</i> amount. The <i>expr</i> can evaluate to an integer, blob or string. If <i>expr</i> evaluates to a string, this function tries to convert it to a signed integer, and if that fails, to a blob. If both fail, it returns an error. The <i>shift</i> must evaluate to something that is convertible to a signed integer. If <i>shift</i> is positive, the shift is to the left; if negative, the shift is to the right. If <i>expr</i> results in a signed integer, the right shift is with sign extension. If <i>expr</i> results in an unsigned integer or blob, a right shift shifts zero bits in on the most significant bits.

Table 24-1 Expression Functions (continued)

Function	Example
Description	
(bit-and <i>arg1 arg2</i>)	(bit-and 00:20 00:ff) returns 00:20
(bit-or <i>arg1 arg2</i>)	(bit-or 00:20 00:ff) returns 00:ff
(bit-xor <i>arg1 arg2</i>)	(bit-xor 00:20 00:ff) returns 00:df
(bit-eqv <i>arg1 arg2</i>)	(bit-andc1 00:20 00:ff) returns 00:df
(bit-andc1 <i>arg1 arg2</i>)	
(bit-andc2 <i>arg1 arg2</i>)	
(bit-orc1 <i>arg1 arg2</i>)	
(bit-orc2 <i>arg1 arg2</i>)	
<p>These bit functions return the result of a bit-wise boolean operation on the two arguments. The data type of the result is a signed integer if both arguments result in either kind of integer, otherwise the result is a blob. The <i>arg1</i> and <i>arg2</i> arguments must evaluate to two integers, two blobs of equal length, or one integer and one blob of length 4. If either argument evaluates to a string, the function tries to convert the string to a signed integer, and if that fails, to a blob. After this conversion, the results must match the criteria mentioned above. If these conditions are not met, it returns an error.</p> <p>Operations with c1 and c2 indicate that the first and second arguments, respectively, are complemented before the operation.</p>	
(bit-not <i>expr</i>)	(bit-not ff:ff) returns 00:00 (bit-not 1) returns 4294967295 (bit-not "hello world") returns an error
<p>The bit-not function returns a value that is the bit-by-bit complement of <i>expr</i>. The datatype of the result is the same as the result of evaluating <i>expr</i> and any subsequent conversions, if the result was a string. The expression must evaluate to an integer of either type, or a blob. If it evaluates to a string, the function tries to convert it to a signed integer; if that fails, to a blob, and if that fails, returns an error.</p>	
(byte <i>arg1</i>)	(byte 150) returns 0x96 (byte 0x96) returns 0x96
<p>The byte function eases creation of one-byte blobs. It returns this blob depending on the data type:</p> <ul style="list-style-type: none"> • sint, uint—Returns a low-order byte of type integer. • blob—Returns the last byte in the blob. • string—Returns the last byte in the string. 	
(comment <i>comment expr1... exprn</i>)	(comment "this is a comment that won't get lost" (request option 82 1))
<p>This function ignores the <i>comment</i> argument, but evaluates the remaining arguments and returns the value of <i>exprn</i>. Use this function for inserting a comment string into an expression.</p>	
(concat <i>arg1 ... argn</i>)	(concat "hello " "world") returns "hello world" (concat -1 "world") returns an error (concat -1 00:01:02) returns the blob ff:ff:ff:ff:00:01:02
<p>This function concatenates the values of the arguments into a string or blob. It ignores null arguments. The first nonnull argument must evaluate to a string or a blob. If it evaluates to an integer, the function converts it to a blob. The datatype of this first nonnull argument (after any conversion) determines the datatype of the result. The function converts all subsequent arguments to the datatype of the result, and if this conversion fails, returns an error.</p>	

Table 24-1 Expression Functions (continued)

Function	Example
Description	
(dotimes (<i>var count-expr</i> [<i>result-expr</i>]) <i>exp1</i> ... <i>expn</i>)	<pre>(let (x y) (setq x 01:02:03) (dotimes (i (length x)) (setq y (concat (substring x i 1) y))) returns null, but after the dotimes y is the reverse of x (dotimes (i 10) (setq i 1)) loops forever!</pre>
The dotimes function creates an environment with a single local integer variable, <i>var</i> , which is initially set to zero, and evaluates <i>exp1</i> through <i>expn</i> . It then increments <i>var</i> by one, and if it is less than <i>count-expr</i> , evaluates <i>exp1</i> through <i>expn</i> again. When <i>var</i> is equal to or greater than <i>count-expr</i> , the function evaluates <i>result-expr</i> and returns it as the result of the entire dotimes . If there is no <i>result-expr</i> , the function returns null.	
The <i>var</i> defines a local variable, and must be an alphabetic name. The <i>count-expr</i> must evaluate to an integer or be convertible to one. The <i>exp1</i> through <i>expn</i> are expressions that can evaluate to any data type. The <i>result-expr</i> is optional, and if it appears, it can evaluate to any data type. When the function evaluates <i>count-expr</i> , <i>var</i> is not bound and cannot appear in <i>count-expr</i> . Alternatively, <i>var</i> is bound for the evaluation of <i>result-expr</i> and has the value of <i>count-expr</i> . If <i>result-expr</i> is omitted, the function returns null.	
Note	Be careful changing the value of <i>var</i> in <i>exp1</i> through <i>expn</i> , because you can easily create an infinite loop (see the example).
(environmentdictionary { get put <i>val</i> delete } <i>attr</i>)	<pre>nrcmd> dhcp set initial-environment-dictionary=first=one,second=2 (environmentdictionary get "first") returns "one" (environmentdictionary get "second") returns "2" (note string 2) (environmentdictionary put "two" "second") returns "second" (environmentdictionary delete "first") returns null</pre>
The environmentdictionary function gets, puts, or deletes a DHCP extension environment dictionary attribute value. The <i>val</i> is the value of the attribute and <i>attr</i> is the attribute name. Both are converted to a string regardless of their initial datatype. The initial environment dictionary cannot be changed, but it can be shadowed (you can redefine something that is in the initial dictionary, but if you remove it, then the original initial value is still there). Note that the get keyword is not optional for a “get.”	
(equal <i>expr1</i> <i>expr2</i> <i>expr3</i>) (equali <i>expr1</i> <i>expr2</i> <i>expr3</i>)	<pre>(equal (request option "dhcp-class-identifier") "docsis") returns the string "docsis" if the value of the option dhcp-class-identifier is a string identical to "docsis" (equali "abc" "ABC") returns "ABC" (equal "abc" "def") returns null (equal "ab" (as-string 61:62)) "this is true") returns "this is true" (equal "ab" 61:62 "this is not true") returns null (equal 01:02:03 01:02:03) returns 01:02:03 (equal (as-blob "ab") 61:62) returns null (equal 1 (to-blob 1)) returns null (equal (null) (request option 20)) returns "*T*" if there is no option 20 in the packet</pre>
The equal function evaluates the equivalency of the result of evaluating <i>expr1</i> and <i>expr2</i> . If they are equal, it returns:	
	<ol style="list-style-type: none"> 1. The value of <i>expr3</i>, if specified, else 2. The value (and datatype, after possible string conversion) of <i>expr2</i>, as long as <i>expr2</i> is not null, else 3. The string “*T*” (since returning null would incorrectly indicate a failed comparison).
If <i>expr1</i> and <i>expr2</i> are not equal, the function returns null.	

Table 24-1 Expression Functions (continued)

Function	Example
Description	
	The arguments can be any datatype. If different, the function converts them to strings (which cannot fail) before comparing them. Note that any string conversion is performed using the equivalent of (to-string ...). Thus, the blob 61:62 is not equal to the "ab" string. Note also that a one-byte blob 01 is not equal to a literal integer 1 (both are converted to strings, and the "01" and "1" strings are not equal). The equali function is identical to the equal function, except that if the comparison is for strings (either because string arguments were used or because the arguments were converted to strings), a case insensitive comparison is used.
(error)	
	The error function returns a "no recovery" error that causes the entire expression evaluation to fail unless there is a try function above the error function evaluation.
(if cond [then else])	(if (equali (substring (request option "dhcp-class-identifier") 0 6) "docsis") (request option 82 1)) returns sub-option 1 of option 82 if the first six characters of the dhcp-class-identifier are "docsis" in any case; otherwise returns null
	The if function evaluates the condition expression <i>cond</i> in an <i>if-then-else</i> sense. If <i>cond</i> evaluates to a value that is nonnull, it returns the result of evaluating the <i>then</i> argument; otherwise it returns the result of evaluating the <i>else</i> argument. Both <i>then</i> and <i>else</i> are optional arguments. If you omit the <i>then</i> and <i>else</i> arguments, the function simply returns the results of evaluating the <i>cond</i> argument. If you omit the <i>else</i> argument and <i>cond</i> evaluates to null, the function returns null. There are no restrictions on the data types of any of the three arguments.
(ip-string blob)	(ip-string 01:02:03:04) returns "1.2.3.4" (ip-string -1) returns "255.255.255.255" (ip-string (as-blob "hello world")) returns "104.101.108.108"
	The ip-string function returns the string representation of the four-byte IP address <i>blob</i> in the form "a.b.c.d". The single argument <i>blob</i> must evaluate to a blob or be convertible into one. If the blob exceeds four bytes, the function uses only the first four to create the IP address string. If the blob has fewer bytes, the function considers the right-most bytes as zero when it creates the IP address string.
(ip6-string blob)	(ip-string (as-blob "hello world")) returns "6865:6c6c:6f20:776f:726c:6400::"
	The ip6-string function returns the string representation of a 16-byte IPv6 address <i>blob</i> in the form "a:b:c:d:e:f:g:h". The single argument <i>blob</i> must evaluate to a blob or be convertible into one. If the blob exceeds 16 bytes, the function uses only the first 16 to create the IPv6 address string. If the blob has fewer bytes, the function considers the right-most bytes as zero when it creates the IPv6 string.
(is-string expr)	(is-string 01:02:03:04) returns null (is-string "hello world") returns "hello world" (is-string 68:65:6c:6c:6f:20:77:6f:72:6c:64) returns the blob
	The is-string function returns the value of <i>expr</i> , if the result of evaluating <i>expr</i> is a string or can be used as a string, this function, otherwise it returns null. That is, if as-string does not return an error, then is-string returns the value of <i>expr</i> .
(length expr)	(length 1) returns 4 (length 01:02:03) returns 3 (length "hello world") returns 11
	The length function returns an integer whose value is the length in bytes of the value of <i>expr</i> . The argument <i>expr</i> can evaluate to any datatype. Integers always have length 4. The length of a string does not include any zero byte that may terminate the string.

Table 24-1 Expression Functions (continued)

Function	Example	Description
(let (<i>var1</i> ... <i>varn</i>) <i>expr1</i> ... <i>exprn</i>)	<pre>(let (x) (setq x (substring (request option "dhcp-class-identifier") 0 6)) (if (equali x "docsis") "client-class-1") (if (equali x "something else") "client-class-2"))</pre>	The let function creates an environment with local variables <i>var1</i> through <i>varn</i> , which are initialized to a null value (you can give them other values by using the setq function). Once the local variables are initialized to null, the function evaluates expressions <i>expr1</i> through <i>exprn</i> in order. It then returns the value of its last expression, <i>exprn</i> . The benefit of this function is that you can use it to calculate a value once, assign it to a local variable, then re-use that value in other expressions without having to recalculate it. Variables are case sensitive.
(log <i>severity</i> <i>expr</i>)		The log function logs the result of converting <i>expr</i> to a string. The <i>severity</i> and <i>expr</i> must be a string and are converted to one if they do not evaluate to one. The <i>severity</i> can also be null; if a string, it must have one of these values: <pre>"debug" "activity" (the default if severity is null) "info" "warning" "error"</pre>
Note		Logging consumes considerable server resources, so limit the number of log function evaluations you put in an expression. Even if “error” severity is logged, the log function does not return an error. This only tags the log message with an error indication. See the error function to return an error as part of a function evaluation.
(mask-blob <i>mask-size</i> <i>length</i>)	<pre>(mask-blob 1 4) yields 80:00:00:00 (mask-blob 4 2) yields f0:00 (mask-blob 31 4) yields ff:ff:ff:fe</pre>	The mask-blob function returns a blob that contains the mask of length <i>mask-size</i> starting from the high-order bit of the blob, with a blob length of <i>length</i> . The <i>mask-size</i> is an expression that evaluates to an integer or must be convertible to one. Likewise the <i>length</i> , which cannot be smaller than the <i>mask-size</i> , but has no fixed limit except that it must be zero or positive. If <i>mask-size</i> is less than zero, it denotes a mask length calculated from the right end of the blob.
(mask-int <i>mask-size</i>)	<pre>(mask-int 1) yields 0x80000000 (mask-int 4) yields 0xf0000000 (mask-int 31) yields 0xfffffffffe (mask-int -1) yields 0x00000001</pre>	The mask-int function returns an integer mask of length <i>mask-size</i> bits starting from the high-order bit of the integer. The <i>mask-size</i> is an expression that evaluates to an integer or must be convertible to one. Any number over 32 is meaningless and is treated as though a value of 32 was used. If <i>mask-size</i> is less than zero, it denotes a mask length calculated from the right end of the integer.
(not <i>expr</i>)	<pre>(not "hello world") return null</pre>	The not function evaluates a string, blob, or integer expression to nonnull if it is null, and null if it is nonnull. The nonnull value returned when the value of <i>expr</i> is null is not guaranteed to remain the same over two calls.
(null [<i>expr1</i> ... <i>exprn</i>])		The null function returns null and does not evaluate any of its arguments.

Table 24-1 Expression Functions (continued)

Function	Example
Description	
(or arg1 ... argn) (pick-first-value arg1 ... argn)	<pre>(or (request option 82 1) (request option 82 2) 01:02:03:04)</pre> <p>returns the value of sub-option 1 in option 82, and if that does not exist, returns the value of sub-option 2, and if that does not exist, returns 01:02:03:04</p>
The or or pick-first-value function evaluates the arguments sequentially. When evaluating an <i>arg</i> returns a nonnull value, it returns the value of that first nonnull argument. Otherwise, it returns the value of the last argument, <i>argn</i> . The datatypes need not be the same.	
(progn arg ... argn)	<pre>(progn (log (null) "I was here") (request option 82 1))</pre>
The progn function evaluates arguments sequentially and returns the value of the last argument, <i>argn</i> .	
(request [get get-blob] [relay [n]] option opt [{instance n} {enterprise-id n} {vendor string}][[subopt {option opt}][{instance n} {enterprise-id n} {vendor string}][[sub-subopt {option opt}][{instance n} {enterprise-id n} {vendor string}][instance-count {index n} count])	<pre>(request option 82)</pre> returns the relay-agent-info option as a blob <pre>(request option 82 1)</pre> returns just the circuit-id (1) suboption <pre>(request option 82 "circuit-id")</pre> is the equivalent <pre>(request option "domain-name-servers")</pre> returns the first IP address from the domain-name-servers option <pre>(request option 6 index 0)</pre> is the equivalent <pre>(request option 6 count)</pre> returns the number of IP addresses <pre>(request get-blob option "dhcp-class-identifier")</pre> returns the value as a blob, not a string <pre>(request option "IA-NA" instance 2 option "IAADDR" instance 3)</pre> returns the third instance of the IA-NA option, and the fourth instance of the IAADDR option encapsulated in the IA-NA option <pre>(request get-blob option "vendor-opts" enterprise-id 1234)</pre> returns a blob of the option data for enterprise-id 1234 <pre>(request option "vendor-opts" enterprise-id 1234 3)</pre> returns suboption 3 from the requested vendor option data
The request option function returns the value of the option from the packet. The keywords are:	
<ul style="list-style-type: none"> • get—Optional and assumed if omitted. • get-blob—Returns the data as a blob, providing direct access to the option bytes. • relay—Applies to IPv6 packets only, otherwise returns an error. Requests a relay option instead of a client option. The <i>n</i> indicates the <i>n</i>th closest relay agent to the client; if omitted, 0 (the relay agent nearest to the client) is assumed. • option—Options are specified with the <i>opt</i> argument, which must evaluate to an integer or a string. If it does not evaluate to one of these, the function does not convert it and returns an error. Valid string values for the <i>opt</i> specifier are the same as those used for extensions. • instance—Selects the (<i>n</i>+1)th instance of the preceding option or suboption. Instances start at 0. • enterprise-id—After an option or suboption and for DHCPv4 and DHCPv6, returns only the data bytes after the given enterprise-id in the packet, instead of the option's entire data. 	
(continued)	

Table 24-1 Expression Functions (continued)

Function	Example
Description	
<ul style="list-style-type: none"> • vendor—After an option or suboption, requests that the vendor’s custom option definition be used for decoding the data in the option. Does not apply to DHCPv6 options. Note that if no definition exists for the specified vendor string, no error is issued and the option’s standard definition is used (or, if none, it is assumed to be a blob). • instance-count—Returns the number of instances of the preceding option or suboption, and is usually used to loop through all instances of it. • index—Selects the (<i>n</i>+1)th value in an option that contains multiple values. For example, index 0 returns the first value and index 1 returns the second value. • count—Returns the number of relevant data items in the preceding option, and is usually used with the index keyword to loop through all data values for an option or suboption. 	

The only string-valued suboption names defined for the *subopt* (suboption) specifier are for the relay-agent-info option (82) and are:

```

1—"circuit-id"
2—"remote-id"
4—"device-class"
5—"subnet-selection"
6—"subscriber-id"
7—"radius-attributes" (which includes the following encapsulated attributes that can be specified
as subsuboptions: 1—"radius-user", 6—"radius-class", 88—"radius-framed-pool-name",
26—"radius-vendor-specific", 27—"radius-session-timeout")
8—"authentication"
9—"v-i-vendor-class"
150—"cisco-subnet-selection"
151—"cisco-vpn-id"
152—"cisco-server-id-override"
181—"vpn-id"
182—"server-id-override"

```

The **request option** function returns a value with a datatype depending on the option requested. This shows how the datatypes in the table correspond to the datatypes returned by the **request** function:

```

blob —> blob
IP address —> 4-byte blob
string —> string
8-bit unsigned integer —> uint
16-bit unsigned integer —> uint
32-bit unsigned integer —> uint
integer —> sint
byte-valued boolean —> sint=1 if true, null if false

```

Table 24-1 Expression Functions (continued)

Function	Example
Description	
(request [get get-blob] [relay [<i>number</i>]] <i>packetfield</i>)	(request get ciaddr) returns the ciaddr if it exists, otherwise returns null (request ciaddr) is the same as (request get ciaddr) (request giaddr)
Valid values for <i>packetfield</i> are:	The request packetfield function returns the value of the named field from the request packet. DHCP request packets contain named fields as well as options in an option area. This form of the request function is used to retrieve specific named fields from the request packet. The relay keyword is described in the earlier, more general request function.
op (blob 1)	
htype (blob 1)	
hlen (blob 1)	The <i>packetfield</i> values defined in RFC 2131 are listed at the left. There are several <i>packetfield</i> values that can be requested which do not appear in exactly these ways in the raw DHCP packet. These take data that appears in the packet and combine it in commonly used ways. In these explanations, the packet contents assumed are:
hops (blob 1)	
xid (uint)	
secs (uint)	
flags (uint)	<i>hlen</i> = 1
ciaddr (blob 4)	<i>htype</i> = 6
yiaddr (blob 4)	<i>chaddr</i> = 01:02:03:04:05:06
siaddr (blob 4)	macaddress-string (string)—Returns the MAC address in <i>hlen,htype,chaddr</i> format (for example, “1,6,01:02:03:04:05:06”)
giaddr (blob 4)	macaddress-blob (blob)—Returns the MAC address in <i>hlen:htype:chaddr</i> format (for example, 01:06:01:02:03:04:05:06)
chaddr (blob <i>hlen</i>)	macaddress-clientid (blob)—Returns a client-id created from the MAC address in the Microsoft <i>htype:chaddr</i> client-id format (for example, 01:01:02:03:04:05:06)
sname (string)	
file (string)	
Valid values for the DHCPv6 <i>packetfield</i> are:	The msg-type packet field for DHCPv6 describes the current relay or client message type, and has the values:
msg-type (uint)	1=SOLICIT, 2=ADVERTISE, 3=REQUEST, 4=CONFIRM,
msg-type-name (string)	5=RENEW, 6=REBIND, 8=RELEASE, 9=-DECLINE, 11=INFORMATION-REQUEST, 12=RELAY-FORWARD
xid (uint)	The msg-type-name packet field returns a string of the message type name. The string value is always uppercase; for example, SOLICIT.
relay-count (uint)	The xid is the 24-bit client transaction ID, and the relay-count is the number of relay messages in the request.
hop-count (uint)	
link-address (blob 16)	If a DHCPv6 packet field is requested from a DHCPv4 packet, an error is returned. The inverse is also true.
peer-address (blob 16)	
(request dump)	
	The request dump function dumps the current request packet to the log file after the function evaluates the expression. Note that not all expression evaluations support the dump keyword, but when it is not supported, it is ignored.

Table 24-1 Expression Functions (continued)

Function	Example
Description	
(search <i>arg1</i> <i>arg2</i> <i>fromend</i>)	<code>(search "test" "this is a test")</code> returns 9 <code>(search "test" "this test test test" "true")</code> returns 15
<p>The search function searches <i>arg1</i> for a subsequence in <i>arg2</i> that exactly matches. If found, it returns the index of the element in <i>arg2</i> where the subsequence begins (unless you use the <i>fromend</i> argument); otherwise it returns null. (If <i>arg1</i> is null, it returns 0; if <i>arg2</i> is null, it returns null.) The function does an implicit as-blob conversion on both arguments. Thus, it compares the actual byte sequences of strings and blobs, and sints and uints become 4-byte blobs for the purpose of comparison.</p> <p>A non-null <i>fromend</i> argument returns the index of the leftmost element of the rightmost matching subsequence.</p>	
(setq <i>var</i> <i>expr</i>)	see the let function for examples
<p>The setq function sets <i>var</i> to the value of <i>expr</i>. You must precede it with the let function.</p>	
(starts-with <i>expr</i> <i>prefix-expr</i>)	<code>(starts-with "abcdefghijklmnp" "abc")</code> returns "abcdefghijklmnp" <code>(starts-with "abcdefgji" "bcd")</code> returns null <code>(starts-with 01:02:03:04:05:06 01:02:03)</code> returns 01:02:03:04:05:06 <code>(starts-with "abcd" (as-string 61:62))</code> returns "abcd" <code>(starts-with "abcd" 61:62)</code> returns null <code>(starts-with "abcd" (to-string 61:62))</code> returns null
<p>The starts-with function returns the value of <i>expr</i> if the <i>prefix-expr</i> value matches the beginning of <i>expr</i>, otherwise null. If <i>prefix-expr</i> is longer than <i>expr</i>, it returns null. The function returns an error if <i>prefix-expr</i> cannot be converted to the same datatype as <i>expr</i> (string or blob), or if <i>expr</i> evaluates to an integer. (See Table 24-2 on page 24-16.)</p>	
(substring <i>expr</i> <i>offset</i> <i>len</i>)	<code>(substring "abcdefg" 0 6)</code> returns bcdefg <code>(substring 01:02:03:04:05:06 3 2)</code> returns 04:05
<p>The substring function returns <i>len</i> bytes of expression <i>expr</i>, starting at <i>offset</i>. The <i>expr</i> can be a string or blob; if an integer, converts to a blob. The result is a string or a blob, or null if any argument evaluates to null. If <i>offset</i> is greater than the length <i>len</i>, the result is null. If <i>offset</i> plus <i>len</i> is data beyond the end of <i>expr</i>, the function returns the rest of the data in <i>expr</i>. If <i>offset</i> is less than zero, the offset is from the end of the data (the last character is index -1, because -0=0, which references the first character). If this references data beyond the beginning of data, the offset is considered to be zero.</p>	
(to-blob <i>expr</i>)	<code>(to-blob 1)</code> returns 00:00:00:01 <code>(to-blob "01:02")</code> returns 01:02 <code>(to-blob 02:03)</code> returns 02:03 <code>(to-blob "this is not in blob format")</code> return an error
<p>The to-blob function converts an expression to a blob. If <i>expr</i> evaluates to a string it must be in "nn:nn:nn" format. This function returns a blob that is the result of converting the string to a blob. If the function cannot convert the string to a blob, it returns an error. If <i>expr</i> evaluates to a blob, it returns that blob. If <i>expr</i> evaluates to an integer, it returns a four-byte blob representing the bytes of the integer in network order. (See Table 24-2 on page 24-16.)</p>	
(to-lower <i>expr</i>)	
<p>The to-lower function takes a string and produces a lowercase string from it. When using the <i>client-lookup-id</i> attribute to calculate a client-specifier to look up a client-entry in the MCD local store (as opposed to LDAP), the resulting string must be lowercase. Use this function to easily make the result of the <i>client-lookup-id</i> a lowercase string. You may or may not wish to use this function when accessing LDAP using the <i>client-lookup-id</i>.</p>	

Table 24-1 Expression Functions (continued)

Function	Example
Description	
(to-sint <i>expr</i>)	<pre>(to-sint "1") returns 1 (to-sint -1) returns -1 (to-sint 00:02) returns 2 (to-sint "00:02") returns an error (to-sint "4294967295") returns an error</pre>
<p>The to-sint function converts an expression to a signed integer. If <i>expr</i> evaluates to a string, it must be in a format that can be converted into a signed integer, else the function returns an error. If <i>expr</i> evaluates to a blob of one to four bytes, the function returns it as a signed integer. If <i>expr</i> evaluates to a blob of more than 4 bytes in length, it returns an error. If <i>expr</i> evaluates to an unsigned integer, it returns a signed integer with the same value, unless the value of the unsigned integer was greater than the largest positive signed integer, in which case it returns an error. If <i>expr</i> evaluates to a signed integer, it returns that value. (See Table 24-2 on page 24-16.)</p>	
(to-uint <i>expr</i>)	<pre>(to-uint "1") returns 1 (to-uint 00:02) returns 2 (to-uint "4294967295") returns 4294967295 (to-uint "00:02") returns an error (to-uint -1) returns an error</pre>
<p>The to-uint function converts an expression to an unsigned integer. If <i>expr</i> evaluates to a string, it must be in a format that can be converted into an unsigned integer, else the function returns an error. If <i>expr</i> evaluates to a blob of one to four bytes, it returns it as an unsigned integer. If <i>expr</i> evaluates to a blob of more than 4 bytes in length, it returns an error. If <i>expr</i> evaluates to a signed integer, it returns an unsigned integer with the same value, unless the value of the signed integer less than zero, in which case it returns an error. If <i>expr</i> evaluates to an unsigned integer, the function returns that value. (See Table 24-2 on page 24-16.)</p>	
(to-string <i>expr</i>)	<pre>(to-string "hello world") returns "hello world" (to-string -1) returns "-1" (to-string 02:04:06) returns "02:04:06"</pre>
<p>The to-string function converts an expression to a string. If <i>expr</i> evaluates to a string, it returns it; if a blob or integer, it returns its printable representation. It never returns an error if <i>expr</i> itself evaluates without error, because every value has a printable representation. (See Table 24-2 on page 24-16.)</p>	
(try <i>expr failure-expr</i>)	<pre>(try (try (expr) (complex-failure-expr)) "string-constant" ensures that the outer try never returns an error (because evaluating "string-constant" cannot fail). (try (error) 01:02:03) always returns 01:02:03 (try 1 01:02:03) always returns 1 (try (request option 82) "failure") never returns "failure" because (request option 82) turns null if there is no option-82 in the packet and does not return an error (try (request option "junk") "failure") returns "failure" because "junk" is not a valid option-name.</pre>
<p>The try function evaluates <i>expr</i> and returns the result of that evaluation if there were no errors encountered during the evaluation. If an error occurs while evaluating <i>expr</i> then:</p> <ul style="list-style-type: none"> • If there is a <i>failure-expr</i> and it evaluates without error, it returns the result of that evaluation as the result of the try function. • If there is a <i>failure-expr</i> and the function encounters an error while evaluating <i>failure-expr</i>, it returns that error. • If there is no <i>failure-expr</i>, the try returns null. 	

Table 24-1 Expression Functions (continued)

Function	Example
Description	
(+ <i>arg1</i> ... <i>argn</i>)	(+ 1 2 3 4) returns 10
(- <i>arg1</i> ... <i>argn</i>)	(- 10 5 2) returns 3
(* <i>arg1</i> ... <i>argn</i>)	(* 3 4 5) returns 60
(/ <i>arg1</i> ... <i>argn</i>)	(/ 20 2 5) returns 2
	(/ 20 0) returns an error

These functions do arithmetic operations on a signed integer or an expression that can be converted to a signed integer. Any argument that cannot be converted to a signed integer (and is not null) returns an error. Any argument that evaluates to null is ignored (except that the first argument for - and / must not evaluate to null). These functions always return signed integers (note that overflow and underflow are currently not caught):

- + sums the arguments; if no arguments, the result is 0.
- - negates the value of a single argument or, if multiple arguments, successively subtracts the values of the remaining ones from the first one; for example, (- 3 4 5) becomes -6.
- * takes the product of the argument values; if no arguments, the result is 1.
- / successively divides the first argument by all of the others; for example, (/ 100 4 5) becomes 5. If any argument other than the first equals 0, an error is returned.

Datatype Conversions

When a function needs an argument of a particular datatype, it tries to convert a value into that datatype. Sometimes this can fail, often causing the entire function to fail. Datatype conversion is also performed by the **to-string**, **to-blob**, **to-sint**, and **to-uint** functions. Whenever a function needs an argument in a specific datatype, it calls the internal version of these externally available functions.

There are also **as-string**, **as-blob**, **as-sint**, and **as-uint** conversion functions, where the data in a value are simply relabeled as the desired datatype, although some checking does go on. The conversion matrix for both function sets appears in [Table 24-2](#).

Table 24-2 Datatype Conversion Functions

Argument Type:	String	Blob	Signed Integer	Unsigned Integer
Function:				
to-string	—	Cannot fail	Cannot fail	Cannot fail
as-string	—	Relabels as string bytes, if printable characters	Converts to a 4-byte blob, then processes it as a blob (which fails except for a few special integers)	Converts to a 4-byte blob, then processes as a blob (which fails except for a few special integers)
to-blob	Must be in the form “01:02:03”	—	Cannot fail; produces a 4-byte blob from the 4 bytes of the integer.	Cannot fail; produces a 4-byte blob from the 4 bytes of the integer.
as-blob	Cannot fail; relabels ASCII characters as blob bytes.	—	Cannot fail; produces 4-byte blob from the 4 bytes of the integer.	Cannot fail; produces 4-byte blob from the 4 bytes of the integer.

Table 24-2 Datatype Conversion Functions (continued)

Argument Type:	String	Blob	Signed Integer	Unsigned Integer
to-sint	Must be in the form <i>n</i> or <i>-n</i> .	1-, 2-, 3-, or 4-byte blobs only.	—	Converts only if it is not too big to fit into a signed integer.
as-sint	Not usually useful; converts a 1-, 2-, 3-, or 4-byte string to a blob and then packs it up into a signed integer.	Not usually useful; converts only 1-, 2-, 3-, or 4-byte blobs.	—	Cannot fail; converts to a signed integer, negative if a larger unsigned integer would fit into a positive signed integer.
to-uint	Must be in the form <i>n</i> .	1-, 2-, 3-, or 4-byte blobs only.	Nonnegative only.	—
as-uint	Not usually useful; converts a 1-, 2-, 3-, or 4-byte string to a blob and then a signed integer.	Not usually useful; converts only 1-, 2-, 3-, or 4-byte blobs.	Cannot fail; converts to an unsigned integer, and a negative signed integer becomes a large unsigned integer.	—

Expressions in the CLI

You must include the expression in a text file if you want to include it with a CLI attribute setting. The default path of this file is the current working directory. Do not enclose the expression in quotes. You can add comment lines prefixed by #, //, or ;. For example:

```
// Expression to set client-class based on remote-id
(if (equal (request option "relay-agent-info" "remote-id") (request chaddr)
          "no-limit" "limit"))
```

The file is read when the command is processed. An example of a command to include this file is:

```
nrcmd> dhcp set client-class-lookup-id=@expressionfile1.txt
```

Expression Examples

These examples provide the maximum support for option 82 processing. They set up clients to limit, those not to limit, and those that exceed configuration limits and should be assigned to an over-limit client-class. There are separate scopes and scope-selection tags for each of the three classes of clients:

- Client-classes—limit, no-limit, and over-limit.
- Scopes—10.0.1.0 (primary), 10.0.2.0 and 10.0.3.0 (secondaries), named for their subnets.
- Scope-selection tags—limit-tag, no-limit-tag, and over-limit-tag. The scopes are named for the address pools that they represent. The selection tags are allocated to the scopes with 10.0.1.0 getting limit-tag, 10.0.2.0 getting no-limit-tag, and 10.0.3.0 getting over-limit-tag.

Limitation Example 1: DOCSIS Cable Modem

The test is to determine whether the device is considered a DOCSIS cable modem, and limit the number of customer devices behind every cable modem. The limitation ID for the limit client-class is the cable modem's MAC address, included in the *remote-id* suboption of the *relay-agent-info* option.

The expression for the *client-class-lookup-id* attribute on the server is:

```
// Expression to set client-class to no-limit or limit based on remote-id
(if (equal (request option "relay-agent-info" "remote-id")
  (request chaddr))
  "no-limit"
  "limit")
```

The above expression indicates that if the contents of the *remote-id* suboption (2) of the *relay-agent-info* option is the same as the *chaddr* of the packet, then the client-class is *no-limit*, otherwise *limit*.

The *limitation-id* expression for the *limit* client-class is:

```
(request option "relay-agent-info" "remote-id")
```

Use this expression in the following steps:

-
- Step 1** Define the scope-selection tags. (You do not need to do this in Network Registrar 6.0 or 6.1.)
 - Step 2** Define the client-classes.
 - Step 3** Define the scopes, their ranges and tags, and if they are primary or secondary. Note the host range for each scope, which is less likely to be misread than if they all have the same host number.
 - Step 4** Define the limitation count. It can go in the *default* policy; if the request does not show a limitation ID, the count is not checked.
 - Step 5** Add an expression in an expression file, *cclookup1.txt*, for the purpose:


```
// Expression to set limitation count based on remote-id
(if (equal (request option "relay-agent-info" "remote-id")
  (request chaddr)) "no-limit" "limit")
```
 - Step 6** Refer to the expression file when setting the *client-class lookup-id* attribute on the server level.
 - Step 7** Add another expression for the limitation ID for the client in a *cclimit1.txt* file:


```
// Expression to set limitation ID based on remote-id
(request option "relay-agent-info" "remote-id")
```
 - Step 8** Refer to this expression file when setting the *limitation-id* attribute for the client-class.
 - Step 9** Reload the server.
-

The result of doing this for a previously unused configuration would be to put the first two DHCP clients with a common *remote-id* option 82 suboption value in the *limit* client-class. The third client with the same value would go in the *over-limit* client-class. There are no limits to the number of devices a subscriber can have in the *no-limit* client-class, because it has no configured limitation ID. Any device with a MAC address equal to the value of the *remote-id* suboption is ignored for the purposes of limitation, and goes in the *no-limit* client class, for which there is no limitation ID configured.

Limitation Example 2: Extended DOCSIS Cable Modem

This example is an extension to the example described in the “[Limitation Example 1: DOCSIS Cable Modem](#)” section on page 24-18. In the latter example, all of the cable modems allowed only two client devices beyond them, since a limitation count of two was defined for the *default* policy. In this example, specific cable-modems are configured to allow a different number of devices to be granted IP addresses from the scopes that use the *limit-tag* scope-selection tag.

In this case, you need to explicitly configure any cable modem with more than two addresses behind it in the client-class database. This requires enabling client-class processing server-wide, so that you can look up the client entry for a cable modem in the Network Registrar or LDAP database. Not finding the cable modem limits the number of devices to two; finding it uses the limitation count from the policy configured for the cable modem.

This example requires just one additional policy, *five*, which allows five devices.

-
- Step 1** Enable client-class processing server-wide.
- Step 2** Create the *five* policy with a limitation count of five devices.
- Step 3** As in the previous example, use an expression to set a limitation ID for the *limit* client-class. Put the limitation ID in a *cclimit2.txt* file, and the lookup ID in a *cclookup2.txt* file:
- ```
cclimit2.txt file:
// Expression to set limitation ID
(request option "relay-agent-info" "remote-id")

cclookup2.txt file:
// Expression to set client-class lookup ID
(concat "1,6," (to-string (request option "relay-agent-info" "remote-id")))
```
- Step 4** Refer to these files when setting the appropriate attributes.
- Step 5** Define some cable modem clients and apply the *five* policy to them.
- Step 6** Reload the server.
- 

## Limitation Example 3: DSL Over Asynchronous Transfer Mode

This example shows how to use expressions to configure Digital Subscriber Line (DSL) access for a subscriber to a service provider using asynchronous transfer mode (ATM) routed bridge encapsulation (RBE). Service providers are increasingly using ATM RBE to configure a DSL subscriber. The DHCP Option 82 support for routed bridge encapsulation feature as of Cisco IOS Release 12.2(2)T enables those service providers to use DHCP to assign IP addresses and option 82 to implement security and IP address assignment policies.

In this scenario, DSL subscribers are identified as individual ATM subinterfaces on a Cisco 7401ASR router. Each customer has their own subinterface in the router and each subinterface has its own virtual channel identifier (VCI) and virtual path identifier (VPI) to identify the next destination of an ATM cell as it passes through ATM switches. The 7401ASR router routes up to a Cisco 7206 gateway router.

---

**Step 1** Set up the DHCP server and interfaces for the router using IOS. This is a typical IOS configuration:

```
Router#ip dhcp-server 170.16.1.2
Router#interface Loopback0
Loopback0(config)#ip address 11.1.1.129 255.255.255.192
Loopback0(config)#exit
Router#interface ATM4/0
ATM4/0(config)#no ip address
ATM4/0(config)#exit
Router#interface ATM4/0.1 point-to-point
ATM4/0.1(config)#ip unnumbered Loopback0
ATM4/0.1(config)#ip helper-address 170.16.1.2
ATM4/0.1(config)#atm route-bridged ip
ATM4/0.1(config)#pvc 88/800
ATM4/0.1(config)#encapsulation aal5snap
ATM4/0.1(config)#exit
Router#interface Ethernet5/1
Ethernet5/1(config)#ip address 170.16.1.1 255.255.0.0
Ethernet5/1(config)#exit
Router#router eigrp 100
eigrp(config)#network 11.0.0.0
eigrp(config)#network 170.16.0.0
eigrp(config)#exit
```

**Step 2** In IOS, enable the system to insert the DHCP option 82 data in forwarded BOOTREQUEST messages to a Cisco IOS DHCP server:

```
Router#ip dhcp relay information option
```

**Step 3** In IOS, specify the IP address of the loopback interface on the DHCP relay agent that is sent to the DHCP server using the option 82 *remote-id* suboption (2):

```
Router#rbe nasip Loopback0
```

**Step 4** In Network Registrar, enable client-class processing server-wide.

**Step 5** Create the *one* policy with a limitation count of one device.

**Step 6** Put the packets in the right client-class. All the packets should be in the *limit* client-class. Create a lookup file containing just the value *limit*, then set the client-class lookup ID. In the *cclookup3.txt* file:

```
// Sets client-class to limit
"limit"
```

**Step 7** Use an expression to ensure that those packets that are limited have the right limitation ID. Put the expression in a file and refer to that file to set the limitation ID. The *substring* function gets the VPI/VCI by extracting bytes 10 through 12 of the option 82 suboption 2 (*remote-id*) data field. In the *cclimit3.txt* file:

```
// Sets limitation ID
(substring (request option 82 2) 9 3)
```

**Step 8** Reload the server.

---

# Debugging Expressions

If you are having trouble with expressions, examine the DHCP log file at server startup. The expression is printed in such a way as to clarify the nesting of functions, and can help in confirming your intentions. Pay special attention to the **equal** function and any datatype conversions of arguments. If the arguments are not the same datatype, they are converted to strings using code similar to the **to-string** function.

You can set various debug levels for expressions by using the *expression-trace-level* attribute for the DHCP server. All executed expressions are traced to the degree set by the attribute. The highest trace level is 10. If you set the level to at least 2, any nonworking expression is retried again at level 10.

The trace levels for *expression-trace-level* are (use the number value):

- **0**—No tracing
- **1**—Failures, including those protected by (**try ...**)
- **2**—Total failure retries (with trace level = 6 for retry)
- **3**—Function calls and returns
- **4**—Function arguments evaluated
- **5**—Print function arguments
- **6**—Datatype conversions (everything)

The trace levels for *expression-configuration-trace-level* are (use the number value):

- **0**—No additional tracing
- **1**—No additional tracing
- **2**—Failure retry (the default)
- **3**—Function definitions
- **4**—Function arguments
- **5**—Variable lookups and literal details
- **6**—Everything

To trace expressions you have trouble configuring, there is also an *expression-configuration-trace-level* attribute that you can set to any level from 1 through 10. If you set the level to at least a 2, any expression that does not configure is retried again with the level set to 6. Gaps in the numbering are to accommodate future level additions.

