Using Expressions

Cisco Prime 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 the active leases of a subscriber, 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 Using Extension Points) are used to modify request or response packets. The expressions described here are also not the same as regex.

- Using Expressions, on page 1
- Entering Expressions, on page 2
- Creating Expressions, on page 4
- Expression Functions, on page 8
- Using Expressions for Options, on page 34
- Using Expressions to Limit IP Addresses Leased to Subscribers, on page 35
- Debugging Expressions, on page 38

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 is supported only for DHCPv4 (not DHCPv6).
- **Creating option values**—See Using Expressions for Options, on page 34.

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.

For DHCPv4, you can also 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. Similar processing is possible for DHCPv6 using the *v6-client-class-lookup-id* and *v6-client-lookup-id* expressions.

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 Creating Expressions, on page 4.

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 KB.

Most expressions that are configured with the CLI are stored in a text file which is then associated with the desired configuration attribute. The default path of this file is the current working directory. You can configure a simple expression directly in the CLI without storing it in a text file. 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\"
```

If you want to use a slightly more extensive example to set the client-class *limitation-id*:

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

Entering this expression directly in the CLI is not possible because of limitations in the CLI's command parsing. You must enter more complex expressions by placing them in a text file and then reference that file in the attribute definition prefixed by the "at" symbol (**@**). For example, if that expression is placed in the cclookup.txt file, the CLI command is:

```
nrcmd> dhcp set client-class-lookup-id=@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,

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


// 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>")
```

The IPv6 version of the previous example (using option numbers) is:

```
// Expression to calculate client-class based on DOCSIS 3.0 cm-mac-address
(try
  (if (equal (request option 17 enterprise-id 4491 36)
             (or (request relay option 17 enterprise-id 4491 1026) "none"))
    "v6-cm-client-class"
    "v6-cpe-client-class")
  "<none>")
```

You can also write the previous expression by substituting option names in place of numbers:

```
// Expression to calculate client-class based on DOCSIS 3.0 cm-mac-address
(try
  (if
    (equal
      (or
        (request option
        "vendor-opts" enterprise-id "dhcp6-cablelabs-config" "device-id")
         (substring (request option "client-linklayer-address") 3 8))
      (or
        (request relay option
          "vendor-opts" enterprise-id "dhcp6-cablelabs-config" "cm-mac-address")
        "none"))
    "v6-cm-client-class"
    "v6-cpe-client-class")
  "<none>")
```

The **or** function in the examples ensures that if the packet was not relayed or if the relay agent did not add the option, then the server assumes the client to be a CPE and not a cable modem (CM).

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 Typical Limitation Scenario would be:

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

// Ends the try function
```

The expression 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 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>")
```

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=@cclookup.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)
```

Expression Syntax

Expressions consist solely of functions and literals. Its syntax is similar to that of Lisp. It follows many of the same rules and uses Lisp functions 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 Expressions Can Fail, on page 6.) 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 the customer premise equipment or PC. Note that both functions and literals are expressions. The previous example shows a function as an expression. For literals, see Literals in Expressions, on page 5.

Expression Datatypes

The datatypes that expressions support are:

- **Blob**—Counted series of bytes, with a recommended maximum length of 1 KB.
- **String**—Counted series of NVT ASCII characters, not terminated by a zero byte, with a recommended maximum 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 IPv4 address is a 4-byte blob, while an IPv6 address is a 16 byte blob. All numbers are in network byte order. See Datatype Conversions, on page 7.

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 can use **(byte 1)** as that will return a blob of 01. Alternatively, you can 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 base 10. 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.

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 the table below.

Note the distinction between **to-string** and **as-string**. For example, let us say that you have data in blob format. You could have this data because of the result of a function evaluation (**request get option**) which retrieves data from a request packet, or as the result of processing blob data with substring. If this data, despite being of blob type, actually represent ASCII string data, you might want to use it as a string. You have two choices of conversions—**as-string** and **to-string**. Which one to choose? If the data consists of ASCII bytes and you want to simply recognize that and essentially reset the type of the data as string, you want to use the **as-string** function. This means that, you are going to use the bytes of the blob "as" a string. The blob 00:01 cannot be converted into a string and it will throw an error if you try. The blob 68:65:6c:6c:6f will successfully convert to a string with **as-string** and yield "hello". On the other hand, if you have a series of bytes that may or may not be ASCII data and you want to represent the data in the blob in a string format, you should use **to-string**. For example, **to-string** will turn a two byte blob consisting of first a 0 then a 1 into the string "00:01".

Table 1: Datatype Conversion Matrix

Function	String	Blob	Signed Integer	Unsigned Integer
as-blob	Cannot fail; relabels ASCII characters as blob bytes.	—	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-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.
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)
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.	—
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.

Function	String	Blob	Signed Integer	Unsigned Integer
to-sint	Must be in the form *n* or *–n*.	1-, 2-, 3-, or 4-byte blobs only.	—	Converts only if it is not too big to fit into a signed integer.
to-string	—	Cannot fail	Cannot fail	Cannot fail
to-uint	Must be in the form *n*.	1-, 2-, 3-, or 4-byte blobs only.	Nonnegative only.	—

Expression Functions

The sections below list the expression functions. Expressions must be enclosed in parentheses.

+, -, *, /, %

Syntax:

(+ *arg1 ... argn***)**

(– *arg1 ... argn***)**

(* *arg1 ... argn***)**

(/ *arg1 ... argn***)**

(% *arg1 arg2***)**

Description:

Arithmetic operations on a signed integer or an expression is convertible to a signed integer. Any argument that cannot convert 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.
- **%** is the modulo arithmetic operator to determine the remainder of the result of the first argument divided by the second one; for example, **(% 12 7)** becomes 5 (12 / 7 = 1 * 7 + 5).

Examples:

(+ 1 2 3 4) returns 10

(- 10 5 2) returns 3

(* 3 4 5) returns 60

(/ 20 2 5) returns 2

(/ 20 0) returns an error

(% 12 7) returns 5 (12/7=1*7+5)

---

and

Syntax:

(and *arg1 ... argn***)**

Description:

Evaluates its arguments in order from left to right. If any argument evaluates to null, it stops evaluating its arguments and returns null. Otherwise, it returns the value of its last argument, *argn*.

Examples:

(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, otherwise it returns null.

---

as-blob

Syntax:

(as-blob *expr***)**

Description:

Treats *expr* as if it were a blob. If *expr* evaluates to a string, the bytes that make up the string become the bytes of the blob that is returned. If *expr* evaluates to a blob, that blob is returned unmodified. If *expr* evaluates to either kind of integer, a 4-byte blob containing the bytes of the integer is returned.

Examples:

(as-blob "hello world") returns the blob 68:65:6c:6c:6f:20:77:6f:72:6c:64

---

as-sint

Syntax:

(as-sint *expr***)**

Description:

Treats *expr* as if it were a signed integer. If *expr* 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 *expr* evaluates to a signed integer, it returns the value unchanged; if an unsigned integer, it returns a signed integer with the same bit value.

Examples:

(as-sint ff:ff:ff:ff) returns -1

(as-sint 2147483648) returns an error

---

as-string

Syntax:

(as-string *expr***)**

Description:

Treats *expr* as if it were a string. If *expr* evaluates to a string, it returns that string. If *expr* 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 *expr* 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.

Examples:

(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.

---

as-uint

Syntax:

(as-uint *expr***)**

Description:

Treats *expr* as if it were an integer. If *expr* 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.

Examples:

(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

---

ash

Syntax:

(ash *expr shift***)**

(lshift *expr shift***)**

Description:

Returns an integer or blob with the bits shifted by the *shift* amount. The *expr* can evaluate to an integer, blob or string. If *expr* 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 *shift* must evaluate to something that is convertible to a signed integer. If *shift* is positive, the shift is to the left; if negative, the shift is to the right. If *expr* results in a signed integer, the right shift is with sign extension. If *expr* results in an unsigned integer or blob, a right shift shifts zero bits in on the most significant bits.

Examples:

(ash 00:01:00 1) returns the blob 00:02:00

(lshift 00:01:00 -1) returns the blob 00:00:80

(ash 1 1) returns the unsigned integer 2

---

bit

Syntax:

(bit-and *arg1 arg2***)**

(bit-andc1 *arg1 arg2***)**

(bit-andc2 *arg1 arg2***)**

(bit-eqv *arg1 arg2***)**

(bit-or *arg1 arg2***)**

(bit-orc1 *arg1 arg2***)**

(bit-orc2 *arg1 arg2***)**

(bit-xor *arg1 arg2***)**

Description:

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 *arg1* and *arg2* 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.

Operations with **c1** and **c2** indicate that the first and second arguments, respectively, are complemented before the operation.

Examples:

(bit-and 00:20 00:ff) returns 00:20

(bit-or 00:20 00:ff) returns 00:ff

(bit-xor 00:20 00:ff) returns 00:df

(bit-andc1 00:20 00:ff) returns 00:df

---

bit-not

Syntax:

(bit-not *expr***)**

Description:

Returns a value that is the bit-by-bit complement of *expr*. 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. The datatype of the result is the same as the result of evaluating *expr* and any subsequent conversions.

Examples:

(bit-not ff:ff) returns 00:00

(bit-not 1) returns 4294967295

(bit-not "hello world") returns an error

---

byte

Syntax:

(byte *arg1***)**

Description:

Eases creation of one-byte blobs. It returns this blob depending on the data type:

- **sint**, **uint**—Returns the low order byte of the integer.
- **blob**—Returns the last byte in the blob.
- **string**—Returns the last byte in the string.

Examples:

(byte 150) returns a blob of 96

(byte 0x96) returns a blob of 96

---

comment

Syntax:

(comment *comment expr1 ... exprn***)**

Description:

Does not evaluate its first argument and returns null if there is only one argument. If there is more than one argument, evaluates arguments *expr1* through *exprn*, and returns the value of *exprn*.

Examples:

(comment "this is a comment that won’t get lost" (request option 82 1))

---

concat

Syntax:

(concat *arg1 ... argn***)**

Description:

Concatenates the values of the arguments into a string or blob (ignoring null arguments). The first argument (*arg1*) must evaluate to a string or a blob; if it evaluates to an integer, the function converts it to a blob. The datatype of *arg1* (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.

Examples:

(concat "hello" "world") returns "helloworld"

(concat -1 "world") returns an error

(concat -1 00:01:02) returns the blob ff:ff:ff:ff:00:01:02

---

datatype

Syntax:

(datatype *expr***)**

Description:

Returns the datatype of the result of the expression (*expr*). If the expression evaluates without an error, returns the datatype as a string, which can be:

- "unset" (internal, considered as null)
- "null"
- "uint"
- "sint"
- "string"
- "blob"

---

dotimes

Syntax:

(dotimes (*var count-expr* [*result-expr*]**)** *exp1 ... expn***)**

Description:

Creates an environment with a single local integer variable, *var*, which is initially set to zero, and evaluates *exp1* through *expn*. It then increments *var* by one, and if it is less than *count-expr*, evaluates *exp1* through *expn* again. When *var* is equal to or greater than *count-expr*, the function evaluates *result-expr* and returns it as the result of the entire **dotimes**. If there is no *result-expr*, the function returns null.

The *var* defines a local variable, and must be an alphabetic name. The *count-expr* must evaluate to an integer or be convertible to one. The *exp1* through *expn* are expressions that can evaluate to any data type. The

result-expr is optional, and if it appears, it can evaluate to any data type. When the function evaluates *count-expr*, *var* is not bound and cannot appear in *count-expr*. Alternatively, *var* is bound for the evaluation of *result-expr* and has the value of *count-expr*. If *result-expr* is omitted, the function returns null.

Note Be careful changing the value of *var* in *exp1* through *expn*, because you can easily create an infinite loop (see the example).

Examples:

(let (x y) (setq x 01:02:03) (dotimes (i (length x)) (setq y (concat (substring x i 1) y)))) returns 03:02:01

(dotimes (i 10) (setq i 1)) loops forever!

---

environmentdictionary

Syntax:

(environmentdictionary {**get** | **put** *val* | **delete**} *attr***)**

Description:

Gets, puts, or deletes a DHCP extension environment dictionary attribute value. The *val* is the value of the attribute and *attr* 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." Also, note that for these examples, the initial-environment-dictionary is used, and while that can be used to "configure" expressions, this function can also be used to communicate with extensions through the environment dictionary that is associated with every request and response pair.

Examples:

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

---

equal, equali

Syntax:

(equal *expr1 expr2 expr3***)**

(equali *expr1 expr2 expr3***)**

Description:

The **equal** function evaluates the equivalency of the result of evaluating *expr1* and *expr2*. If they are equal, it returns:

1. The value of *expr3*, if specified, else
2. The value (and datatype, after possible string conversion) of *expr2*, as long as *expr2* is not null, else
3. The string "*T*" (since returning null would incorrectly indicate a failed comparison).

If *expr1* and *expr2* are not equal, the function returns null.

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.

Examples:

(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 61:62

(equal 1 (to-blob 1)) returns null

(equal (null) (request option 20)) returns "*T*" if there is no option 20 in the packet

---

error

Syntax:

(error)

Description:

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

Syntax:

(if *cond* [*then else*]**)**

Description:

Evaluates the condition expression *cond* in an *if-then-else* sense. If *cond* evaluates to a value that is nonnull, it returns the result of evaluating the *then* argument; otherwise it returns the result of evaluating the *else* argument. Both *then* and *else* are optional arguments. If you omit the *then* and *else* arguments, the function simply returns the results of evaluating the *cond* argument. If you omit the *else* argument and *cond* evaluates to null, the function returns null. There are no restrictions on the data types of any of the three arguments.

Examples:

```
(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.

---

ip-string

Syntax:

(ip-string *blob***)**

Description:

Returns the string representation of the four-byte IP address *blob* in the form *"a.b.c.d"*. The single argument *blob* 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.

Examples:

(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"

---

ip6-string

Syntax:

(ip6-string *blob***)**

Description:

Returns the string representation of a 16-byte IPv6 address *blob* in the form *"a:b:c:d:e:f:g:h"*. The single argument *blob* 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.

Note Since there is more than one acceptable way to represent an IPv6 address as a string, comparing the string format of IPv6 addresses is likely to yield inconsistent results. It is best to compare IPv6 addresses as blob values, where no ambiguity exists in the representation of the addresses. See **to-ip6**, if you already have a string formatted IPv6 address.

Examples:

(ip6-string (as-blob "hello world")) returns "6865:6c6c:6f20:776f:726c:6400::"

---

is-string

Syntax:

(is-string *expr***)**

Description:

Returns the value of *expr*, if the result of evaluating *expr* is a string or can be used as a string, otherwise it returns null. That is, if **as-string** does not return an error, then **is-string** returns the value of *expr*.

Examples:

(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 68:65:6c:6c:6f:20:77:6f:72:6c:64

---

length

Syntax:

(length *expr***)**

Description:

Returns an integer whose value is the length, in bytes, of the value of *expr*. The argument *expr* 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.

Examples:

(length 1) returns 4

(length 01:02:03) returns 3

(length "hello world") returns 11

---

let

Syntax:

(let (*var1 ... varn***)** *expr1 ... expn***)**

Description:

Creates an environment with local variables *var1* through *varn*, 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 *expr1* through *exprn* in order. It then returns the value of its last expression, *exprn*. The benefit of this function is that you can use it to calculate a value once, assign it to a local variable, then reuse that value in other expressions without having to recalculate it. Variables are case-sensitive.

Examples:

```
(let (x)
  (setq x (substring (request option "dhcp-class-identifier") 0 6))
  (or (if (equali x "docsis") "client-class-1")
      (if (equali x "something else") "client-class-2")))
```

---

log

Syntax:

(log *severity expr***)**

Description:

Logs the result of converting *expr* to a string. The *severity* and *expr* must be a string and are converted to one if they do not evaluate to one. The *severity* can also be null; if a string, it must have one of these values:

- "debug"
- "activity" (the default if severity is null)
- "info"
- "warning"
- "error"

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

Syntax:

(mask-blob *mask-size length***)**

Description:

Returns a blob that contains the mask of length *mask-size* starting from the high-order bit of the blob, with a blob length of *length*. The *mask-size* is an expression that evaluates to an integer or must be convertible to one. Likewise the *length*, which cannot be smaller than the *mask-size*, but has no fixed limit except that it must be zero or positive. If *mask-size* is less than zero, it denotes a mask length calculated from the right end of the blob.

Examples:

(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

(mask-blob -1 4) yields 00:00:00:01

---

mask-int

Syntax:

(mask-int *mask-size***)**

Description:

Returns an integer that contains a mask of mask-size, starting from the high-order bit of the integer. The mask-size is an expression that evaluates to an integer or must be convertible to one. If mask-size is less than zero, it denotes a mask length calculated from the right end of the integer.

Examples:

(mask-int 1) yields 0x80000000

(mask-int 4) yields 0xf0000000

(mask-int 31) yields 0xfffffffe

(mask-int -1) yields 0x00000001

---

not

Syntax:

(not *expr***)**

Description:

expr is an expression that can evaluate to a string, a blob, or an integer. If the result of that evaluation is non-null, then null is returned. If the result of that evaluation is null, then a nonnull value is returned. The nonnull value returned when the value of *expr* is null is not guaranteed to remain the same over two calls.

Examples:

(not "hello world") returns null

---

null

Syntax:

(null [*expr1 ... exprn*]**)**

Description:

Returns null and does not evaluate any of its arguments.

---

or, pick-first-value

Syntax:

(or *arg1... argn***)**

(pick-first-value *arg1... argn***)**

Description:

Evaluates the arguments sequentially. When the evaluation of an argument returns a nonnull value, that value is returned. No other arguments are evaluated after one argument returns a nonnull value. Otherwise, returns the value of the last argument, *argn*. The datatypes need not be the same.

Examples:

```
(or
  (request option 82 1)
  (request option 82 2)
  01:02:03:04)
```

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.

---

parse

Syntax:

(parse *expr1 expr2***)**

Description:

Returns the blob result of parsing the string *expr1* parsed as the data type specified in *expr2*. If *expr1* is not a string, it is converted to a string. *expr2* must be one of the AT_* data types (either as a string or its numeric value) supported by Cisco Prime Network Registrar (see Option Validation Types).

This function was introduced in Cisco Prime Network Registrar 11.0.

Examples:

(parse 1234 "AT_INT") returns d2:04:00:00

(parse "cisco.com" "AT_DNSNAME") returns 05:63:69:73:63:67:03:63:6f:6d:00

---

progn, return-last

Syntax:

(progn *arg ... argn***)**

(return-last *arg ... argn***)**

Description:

Evaluates arguments sequentially and returns the value of the last argument, *argn*.

Examples:

```
(progn
  (log (null) "I was here")
  (request option 82 1))

(return-last
  (log (null) "I was here")
  (request option 82 1))
```

---

regex

Syntax:

(regex *expr1 expr2 var1... varn***)**

(regex *expr1 expr2***)**

Description:

Searches for sub-strings, matching with regular-expression pattern (*expr1*), in specified target-string (*expr2*) and sets them to specified variables *var1*, *var2*, or *varn*. That means, first sub-string, matching with regular-expression pattern (*expr1*), in specified target-string (*expr2*), will be set to *var1*, second sub-string will be set to *var2*, and so on. You must precede it with the **let** function when specifying variables. This function can also be used without variables, in this case, it returns first sub-string matching with regular-expression pattern (*expr1*), in specified target-string (*expr2*).

As regular-expression pattern matching works only with strings, both patterns (*expr1*) and target-string (*expr2*) must be strings. If they are not, you should use **as-string** function as used in example below.

Examples:

(regex "[H][a-z]+" "Hello World") returns "Hello".

```
(let (x y z)
  (regex "[H][a-z]+" "Hello Hi World" x y z))
```

will set x="Hello", y="Hi", z=null, and return "Hello".

If you wished, you could put additional expressions after the **regex** inside the **let** to operate on x and y.

---

request

Syntax:

(request [**get** | **get-blob**] [**relay** [*number*]] *packetfield***)**

Description:

Valid values for the DHCPv4 *packetfield* are:

op (blob 1)

htype (blob 1)

hlen (blob 1)

hops (blob 1)

xid (uint)

secs (uint)

flags (uint)

ciaddr (blob 4)

yiaddr (blob 4)

siaddr (blob 4)

giaddr (blob 4)

chaddr (blob *hlen*)

sname (string)

file (string)

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 **request option** function.

The *packetfield* values defined in RFC 2131 are listed above. There are several *packetfield* 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:

hlen = 1 *htype* = 6 *chaddr* = 01:02:03:04:05:06

macaddress-string (string)—Returns the MAC address in *hlen*, *htype*, *chaddr* format (for example, “1,6,01:02:03:04:05:06”)

macaddress-blob (blob)—Returns the MAC address in *hlen*:*htype*:*chaddr* format (for example, 01:06:01:02:03:04:05:06)

macaddress-clientid (blob)—Returns a client-id created from the MAC address in the Microsoft *htype* :*chaddr* client-id format (for example, 01:01:02:03:04:05:06)

Valid values for the DHCPv6 *packetfield* are:

msg-type (uint)

msg-type-name (string)

xid (uint)

relay-count (uint)

hop-count (uint)

link-address (blob 16)

peer-address (blob 16)

The **msg-type** packet field for DHCPv6 describes the current relay or client message type, and has the values:

1=SOLICIT, 2=ADVERTISE, 3=REQUEST, 4=CONFIRM, 5=RENEW, 6=REBIND, 8=RELEASE, 9=-DECLINE, 11=INFORMATION-REQUEST, 12=RELAY-FORWARD

The **msg-type-name** packet field returns a string of the message type name. The string value is always uppercase; for example, SOLICIT.

The **xid** is the 24-bit client transaction ID, and the **relay-count** is the number of relay messages in the request.

If a DHCPv6 packet field is requested from a DHCPv4 packet, an error is returned. The inverse is also true.

Examples:

(request get ciaddr) returns the ciaddr if it exists, otherwise returns null

(request ciaddr) is the same as **(request get ciaddr)**

(request giaddr) returns the giaddr if it is non-zero, otherwise returns null.

---

request dump

Syntax:

(request dump)

Description:

Dumps the current request packet to the log file. Note that not all expression evaluations support the **dump** keyword, and when unsupported, it is ignored.

---

request option

Syntax:

(request [get | get-blob] *option-request***)**

where *option-request* is:

1. An optional relay message selector for IPv6 - **relay** [*n*]
2. One or more option clauses (more than one is only supported for IPv6) - **option** *name* | *id* [**vendor** *name* | **enterprise-id** *name* | *id*] [**instance** *n*]
3. Followed by zero or more suboption clauses - *name* | *id* [**vendor** *name* | **enterprise-id** *name* | *id*] [**instance** *n*]
4. Followed by an optional clause - [**instance-count** | **count** | **index** *n*]

Description:

Returns the value of the option from the packet. The keywords are:

- **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 *n* indicates the *n* th closest relay agent to the client; if omitted, 0 (the relay agent nearest to the client) is assumed.
- **option**— Options (and suboptions) are specified with an *id* or *name* 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 name specifier are the same as those used for extensions.
- **enterprise-id**—After an option or suboption, selects the instance of the option or suboption with the specified enterprise-id. The enterprise-id can be specified as an *id* or *name* argument, which must evaluate to an integer or string.
- **vendor**—After an option or suboption, requests that the vendor 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 standard definition of an option is used (or, if none, it is assumed to be a blob).
- **instance**—Selects the *n* th instance of the preceding option or suboption. Instances start at 0. (You cannot use the instance and instance-count together in a single request function.)
- **instance-count**—Returns the number of instances of the preceding option or suboption, and is usually used to loop through all instances of it. Returns 0 if the option or suboption does not exist.
- **index**—Selects the *n* th value in an option that contains multiple values (that is, array of addresses or integer values). Indexes start at 0. 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 listed in the **DHCPv4 and BOOTP Options** table of the Decoded DHCP Packet Data Items section.

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:

Table 2: Datatypes Returned by the request Function

Option Data Type	Returned Data Type
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

Examples:

(request option 82) returns the relay-agent-info option as a blob

(request option 82 1) returns just the circuit-id (1) suboption

(request option 82 "circuit-id") is the equivalent **(request option 82 1)**

(request option "domain-name- servers") returns the first IP address from the domain-name-servers option

(request option 6 index 0) is the equivalent **(request option 6 count)** returns the number of IP addresses

(request get-blob option "dhcp-class-identifier") returns the value as a blob, not a string

(request option "IA-NA" instance 2 option "IAADDR" instance 3) returns the third instance of the IA-NA option, and the fourth instance of the IAADDR option encapsulated in the IA-NA option

(request get-blob option "vendor-opts" enterprise-id 1234) returns a blob of the option data for enterprise-id 1234

(request option "vendor-opts" enterprise-id 1234 3) returns suboption 3 from the requested vendor option data

DHCPv6 Option 16 Vendor-Class (contains length delimited fields):

Data in the DHCPv6 Message:

```
00:10:00:11:00:00:00:7b:00:04:01:02:03:04:00:05:68:65:6c:6c:6f
^   ^ ^   ^ ^         ^ ^               ^ ^                  ^
|   | |   | |         | +--- field 0 ---+ +-- field 1 -------+
|   | |   | |         |
|   | |   | +---------+ enterprise-id 123(10)
|   | +---+ length 17
+---+ Option 16 Vendor-Class
```

(request option 16 enterprise-id 123) -> Type: blob Value: '01:02:03:04'

(request option 16 enterprise-id 456) -> Type: unset Value: 'null'

(request get-blob option 16 enterprise-id 123) -> Type: blob Value: '00:00:00:7b:00:04:01:02:03:04:00:05:68:65:6c:6c:6f'

(request option 16 enterprise-id 123 index 0) -> Type: blob Value: '01:02:03:04'

(request option 16 enterprise-id 123 index 1) -> Type: blob Value: '68:65:6c:6c:6f'

Note DHCPv6 Option 15, User-Class, operates identically.

DHCPv6 Option 17 Vendor Opts (contains sub-options):

Data in the DHCPv6 Message:

```
00:11:00:12:00:00:01:c8:00:01:00:04:0a:0b:0c:0d:00:05:00:02:01:02
^   ^ ^   ^ ^         ^ ^                     ^ ^              ^
|   | |   | |         | +---- suboption 1 ----+ +- suboption 5 -+
|   | |   | |         |
|   | |   | +---------+ enterprise-id 456(10),1c8(16)
|   | +---+ length 18
+---+ Option 17 Vendor-Opts
```

(request option 17 enterprise-id 456) -> Type: blob Value: '00:00:01:c8:00:01:00:04:0a:0b:0c:0d:00:05:00:02:01:02'

(request option 17 enterprise-id 0x1c8) -> Type: blob Value: '00:00:01:c8:00:01:00:04:0a:0b:0c:0d:00:05:00:02:01:02'

(request option 17 enterprise-id 123) -> Type: unset Value: 'null'

(request option 17 enterprise-id 456 index 0) -> Type: blob Value: '00:00:01:c8:00:01:00:04:0a:0b:0c:0d:00:05:00:02:01:02'

(request option 17 enterprise-id 456 1) -> Type: blob Value: '0a:0b:0c:0d'

(request option 17 enterprise-id 456 2) -> Type: unset Value: 'null'

(request option 17 enterprise-id 456 5) -> Type: blob Value: '01:02'

---

requestdictionary

Syntax:

(requestdictionary {**get** | **put** *val* | **delete**} *attr***)**

Description:

Gets, puts, or deletes a DHCP extension request dictionary attribute value. *val* is the value of the attribute and *attr* is the attribute name. Both are converted to a string regardless of their initial datatype. Note that the **get** keyword is not optional for a "get."

---

response

Syntax:

(response [**get** | **get-blob**] [**relay** [*number*]] *packetfield***)**

Description:

Returns the value of the named *packefield* from the response packet. The description and valid values are identical to those for the **request** *packetfield* function.

---

response dump

Syntax:

(response dump)

Description:

Dumps the current response packet to the log file. Note that not all expression evaluations support the **dump** keyword, and when unsupported, it is ignored.

---

response option

Syntax:

(response [get | get-blob] *option-request***)**

where *option-request* is:

1. An optional relay message selector for IPv6 - **relay** [*n*]
2. One or more option clauses (more than one is only supported for IPv6) - **option** *name* | *id* [**vendor** *name* | **enterprise-id** *name* | *id*] [**instance** *n*]
3. Followed by zero or more suboption clauses - *name* | *id* [**vendor** *name* | **enterprise-id** *name* | *id*] [**instance** *n*]
4. Followed by an optional clause - [**instance-count** | **count** | **index** *n*]

Description:

Returns the value of the option from the packet. The keywords are identical to those for the **request** function.

---

responsedictionary

Syntax:

(responsedictionary {**get** | **put** *val* | **delete**} *attr***)**

Description:

Gets, puts, or deletes a DHCP extension response dictionary attribute value. The *val* is the value of the attribute and *attr* is the attribute name. Both are converted to a string regardless of their initial datatype. Note that the **get** keyword is not optional for a "get."

---

search

Syntax:

(search *arg1 arg2 fromend***)**

Description:

Searches the bytes which make up the value of *arg2* for a subsequence of bytes that exactly matches the sequence of bytes in *arg1*. If found, it returns the index of the element in *arg2* where the subsequence begins (unless you set the *fromend* argument to "true" or some other arbitrary nonnull value); otherwise it returns null. (If *arg1* is null, it returns 0; if *arg2* 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.

A nonnull *fromend* argument returns the index of the leftmost element of the rightmost matching subsequence.

Examples:

(search "test" "this is a test") returns 10

(search "test" "this test test test" "true") returns 15

setq

Syntax:

(setq *var expr***)**

Description:

Only valid within the **let** function. *var* must be one of the *var1* through *varn* local variables defined in the enclosing **let** function.

Examples:

See the **let** function for examples

starts-with

Syntax:

(starts-with *expr prefix-expr***)**

Description:

Returns the value of *expr* if the *prefix-expr* value matches the beginning of *expr*, otherwise null. If *prefix-expr* is longer than *expr*, it returns null. The function returns an error if *prefix-expr* cannot be converted to the same datatype as *expr* (string or blob), or if *expr* evaluates to an integer.

Examples:

(starts-with "abcdefghijklmnop" "abc") returns "abcdefghijklmnop"

(starts-with "abcdefgji" "bcd") returns null

(starts-with 01:02:03:04:05:06 01:02:03) returns 01:02:03:04:05:06

(starts-with "abcd" (as-string 61:62)) returns "abcd"

(starts-with "abcd" 61:62) returns null

(starts-with "abcd" (to-string 61:62)) returns null

substring

Syntax:

(substring *expr offset len***)**

Description:

Returns *len* bytes of expression *expr*, starting at *offset*. The *expr* 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:

- *offset* is greater than the length *len*, the result is null.
- *offset* plus *len* is data beyond the end of *expr*, the function returns the rest of the data in *expr*.

- *offset* 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).
- This references data beyond the beginning of data, the offset is considered to be zero.

Examples:

(substring "abcdefg" 1 6) returns "bcdefg".

(substring 01:02:03:04:05:06 3 2) returns 04:05.

---

synthesize-host-name

Syntax:

(synthesize-host-name *method namestem***)**

Description:

Generates a hostname based on the configured method (if none is specified), or the specified *method* and *namestem*.

The valid methods for the *method* argument depend on whether a DHCPv4 or DHCPv6 request is being processed. For DHCPv4, the valid methods are: **default** or one of the v4-synthetic-name-generator enumeration values of: **address**, **client-id**, or **hashed-client-id**. For DHCPv6, the valid methods are: **default** or one of the v6-synthetic-name-generator enumeration values of: **duid**, **hashed-duid**, **cablelabs-device-id**, or **cablelabs-cm-mac-addr**. For more information on these enumeration methods, see Generating Synthetic Names in DHCPv4 and DHCPv6.

The *namestem* argument specifies the *synthetic-name-stem* value of the DNS update configuration (see Creating DNS Update Configurations).

Examples:

(synthesize-host-name) returns "dhcp-rhfxxi5pkjp6o"

(synthesize-host-name "duid" "test") returns "test-00030001010203040506"

(synthesize-host-name "client-id" "test") returns "test-00030001010203040506"

---

to-blob

Syntax:

(to-blob *expr***)**

Description:

Converts an expression to a blob. If:

- *expr* 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.
- *expr* evaluates to a blob, it returns that blob.
- *expr* evaluates to an integer, it returns a four-byte blob representing the bytes of the integer in network order. (See Datatype Conversions, on page 7.)

Examples:

(to-blob 1) returns 00:00:00:01

(to-blob "01:02") returns 01:02

(to-blob 02:03) returns 02:03

---

to-ip, to-ip6

Syntax:

(to-ip *expr***)**

(to-ip6 *expr***)**

Description:

Converts an expression as string, blob, or integer to an IP address. If:

- A string, it must be in dotted decimal IP address format for IPv4 or colon-formatted format for IPv6. Returns the blob IP address determined by parsing the string into an IP address.
- The result is a blob, it returns the first 4 bytes for (to-ip ...) and the first 16 bytes for (to-ip6 ...). If the blob is less than the 4 bytes for to-ip or 16 bytes for to-ip6, it pads the argument blob with zero bytes in the high order bytes.
- The result is an integer, it converts the integer (of either type) into a blob. Because the integers and blobs are in network order, no order change is required.

---

to-lower

Syntax:

(to-lower *expr***)**

Description:

Takes a string and produces a lowercase string from it. When using the *client-lookup-id* attribute to calculate a client-specifier to look up a client-entry in the CNRDB local store (as opposed to LDAP), the resulting string must be lowercase. Use this function to easily make the result of the *client-lookup-id* a lowercase string. You may or may not want to use this function when accessing LDAP using the *client-lookup-id*.

---

to-sint

Syntax:

(to-sint *expr***)**

Description:

Converts an expression to a signed integer.

If *expr* 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:

- *expr* evaluates to a blob of one to four bytes, the function returns it as a signed integer.
- *expr* evaluates to a blob of more than 4 bytes in length, it returns an error.
- *expr* 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.
- *expr* evaluates to a signed integer, it returns that value.

Examples:

(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 2147483647

---

to-string

Syntax:

(to-string *expr***)**

Description:

Converts an expression to a string. If *expr* evaluates to a string, it returns it; if a blob or integer, it returns its printable representation. It never returns an error if *expr* itself evaluates without error, because every value has a printable representation.

Examples:

(to-string "hello world") returns "hello world"

(to-string -1) returns "-1"

(to-string 02:04:06) returns "02:04:06"

---

to-uint

Syntax:

(to-uint *expr*)

Description:

Converts an expression to an unsigned integer. If

- *expr* evaluates to a string, it must be in a format that can be converted into an unsigned integer, else the function returns an error.
- *expr* evaluates to a blob of one to four bytes, it returns it as an unsigned integer.

- *expr* evaluates to a blob of more than 4 bytes in length, it returns an error.
- *expr* 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.
- *expr* evaluates to an unsigned integer, the function returns that value.

Examples:

(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

---

translate

Syntax:

(translate *expr search replace***)**

Description:

Takes as an argument an expression that evalutes to a sequence of bytes (either a string or a blob), and replaces various characters or bytes that appear in *search* with corresponding values (in the same position) in *replace*. If:

- *expr* is a string or blob, the value is left as it is, otherwise it is forced to be a string. If, after processing, *expr* is a string, *search* and *replace* must be strings.
- *expr* is a blob, both *search* and *replace* must also be blobs.
- *replace* is shorter than *search*, the bytes or characters in *search* that do not have corresponding bytes or characters in *replace* are dropped from the output.
- *replace* does not appear, all the bytes or characters in *search* are removed from *expr*.

Examples:

(translate "Hello apple and eve" "abcdef" "123456") returns "H5llo 1ppl5 1n4 5v5"

(translate "a&b$c%d" "%$&") returns "abcd"

---

try

Syntax:

(try *expr failure-expr***)**

Description:

Evaluates *expr* and returns the result of that evaluation if there were no errors encountered during the evaluation. If an error occurs while evaluating *expr* then:

- If there is a *failure-expr* and it evaluates without error, it returns the result of that evaluation as the result of the **try** function.

- If there is a *failure-expr* and the function encounters an error while evaluating *failure-expr*, it returns that error.
- If there is no *failure-expr*, the **try** returns null.

Examples:

(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.

---

unparse

Syntax:

(unparse *expr1 expr2* [*expr3*]**)**

Description:

Returns the string result of unparsing the blob *expr1* as the data type specified in *expr2*, perhaps modified as specified by *expr3*. If *expr1* is not a blob, it is converted to a blob. *expr2* must be one of the AT_* data types (either as a string or its numeric value) supported by Cisco Prime Network Registrar (see Option Validation Types). *expr3* is optional and may have a value of "none", "alternate", or "feature", and any action depends on *expr2*. For example, for the AT_BOOL type, "feature" will return either "enabled" or "disabled", "alternate" will return either "on" or "off", and "none" (or no expr3) will return either "true" or "false".

This function was introduced in Cisco Prime Network Registrar 11.0.

Examples:

(unparse 00 "AT_BOOL" "feature") returns disabled

(unparse 05:63:69:73:63:67:03:63:6f:6d:00 "AT_DNSNAME") returns "cisco.com."

---

validate-host-name

Syntax:

(validate-host-name *hostname***)**

Description:

Takes the *hostname* string and returns a validated hostname, which can be the same as the input *hostname* or modified as follows:

- Space and underscore characters mapped to a hyphen.
- Invalid hostname characters removed. Valid characters are A-Z, a-z, 0-9, and hyphen.
- Null labels removed (“..” changed to “.”).

- Each label in the hostname truncated to 63 characters.

Examples:

(validate-host-name "a b c d e f") returns "a-b-c-d-e-f"

(validate-host-name "_a_b_c_d_e_f_") returns "a-b-c-d-e-f"

(validate-host-name "abcdef") returns "abcdef"

(validate-host-name "a&b*c#d@!e()f") returns "abcdef"

Using Expressions for Options

Starting with Cisco Prime Network Registrar 11.0, you can use expressions to return values for options (DHCPv4 and DHCPv6).

Note the following while using expressions for options:

- An option instance can either have a fixed value or an expression; not both (though an expression can return a fixed value).
- An option instance that is an expression is evaluated whenever that option is added to the response for a client request.
- An option instance that is an expression is NOT evaluated (and not returned) for a lease query (unitary, bulk, active). The reason for this is that the context for evaluating the expression is not available.
- An expression for an option must return one of the following:
  - A null value—In this case, the option is not added to the response.
  - The value <none> (case insensitive)—In this case, the option is not added to the response.
  - A blob value—In this case, the value is returned as this option. This must be the full option data—for vendor options (such as DHCPv4 option 125 and DHCPv6 option 17), this must include the enterprise-id as the first 4 bytes.
  - A string value—In this case, the value is parsed based on the option's definition and the parsed value is returned. If parsing fails, the option is not added to the response.

  Note that the server does NOT continue searching the policy hierarchy for other instances of the option after evaluating the expression regardless of the result.
- The expression trace settings apply to option expressions.
- The options are added to the response in an unpredictable order and thus, options that are expressions and want to use the value of other options in the response dictionary are NOT recommended as they can produce unpredictable results.
- Round-robining of an option's values can be used with options that are expressions—the expression's resulting values are round-robined.
- An option that is an expression cannot generate 0-length option values, as a null value causes the option not to be added to the response.

Note The DHCPv4 options, *dhcp-lease-time* (51), *dhcp-renewal-time* (58), and *dhcp-rebinding-time* (59) are NOT supported with expressions. These must be configured with a value; the DHCP server ignores the option if configured with an expression.

In the CLI, the **policy** help contains details on how to configure an option instance as an expression.

Using Expressions to Limit IP Addresses Leased to Subscribers

These examples 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 selection tags for each of the three classes of clients. These examples assume the following Cisco Prime Network Registrar configuration environment (which will certainly differ from any real environment and is used just for illustration).

- **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.
- **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 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 client-classes.

Step 2 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 3 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 4 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 5 Refer to the expression file when setting the *client-class lookup-id* attribute on the server level.

Step 6 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 7 Refer to this expression file when setting the *limitation-id* attribute for the client-class.

Step 8 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 Limitation Example 1: DOCSIS Cable Modem, on page 35. 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* 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 Cisco Prime 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 Cisco Prime 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. All expressions are printed in such a way as to clarify the nesting of functions, and can help in confirming your intentions. In particular, you can copy the expression printed in the log file and paste it into an editor. Once you remove the characters from the beginning of each line, the expression that results will input correctly (and will be much easier to read and modify). 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 failing 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)

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. 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