Pan Language Manual

Charles Loomis

Abstract

The pan configuration language allows the definition of machine configuration information and an associated schema with a simple, human-accessible syntax. A pan language compiler transforms the configuration information contained within a set of pan templates to a machine-friendly XML format. This manual describes the pan language.

Table of Contents

1. Introduction 2
2. Templates 2
    2.1. Object Templates 3
    2.2. Ordinary Templates 3
    2.3. Declaration Templates 3
    2.4. Unique Templates 3
    2.5. Structure Templates 3
3. Statements 3
    3.1. assignment 3
    3.2. include 4
    3.3. function 4
    3.4. type 4
    3.5. variable 4
    3.6. bind 5
    3.7. valid 5
4. Types 5
    4.1. Primitive Types 6
    4.2. User-Defined Types 6
5. Functions 7
    5.1. User-Defined Functions 7
    5.2. Built-In Functions 8
6. Data Manipulation Language 11
    6.1. Literals 11
    6.2. Variables 14
    6.3. Operators 14
    6.4. Flow Control 15
7. Other Features 16
    7.1. Annotations 16
A. Built-In Function Reference 18
    append 19
    base64_decode 21
    base64_encode 22
    clone 23
    create 24
    debug 25
    delete 26
    deprecated 27
    error 28
    escape 29
    exists 30
    first 31
    format 32
    if_exists 33

index 34
is_boolean 36
is_defined 37
is_double 38
is_list 39
is_long 40
is_nlist 41
is_null 42
is_number 43
is_property 44
is_resource 45
is_string 46
key 47
length 48
list 49
match 50
matches 51
merge 52
nlist 53
next 54
path_exists 55
prepend 56
replace 58
return 59
splice 60
split 61
substr 62
to_boolean 63
to_double 64
to_long 65
to_lowercase 66
to_string 67
to_uppercase 68
traceback 69
unescape 70
value 71

1. Introduction

The pan language is used within the Quattor toolkit to define the desired configuration for one or more machines. The language is primarily a declarative language where elements in a hierarchical tree are set to particular values. The pan syntax is human-friendly and fairly simple, yet allows system administrators to simultaneously set configuration values, define an overall configuration schema, and validate the final configuration against the schema.

2. Templates

A machine configuration is defined by a set of files, called templates, written in the pan language. These templates define simultaneously the configuration parameters, the configuration schema, and validation functions. Each template is named and is contained in a file having the same name.

The syntax of a template file is simple:

```
[ object | declaration | unique | structure ] template template-name;
[ statement ... ]
```

These files may contain comments that start with the hash sign ('#') and terminate with the next new line or end of file. There are five different types of templates that are identified by the template modifier.

2.1. Object Templates

An object template is declared via the `object` modifier. Each object template is associated with a machine profile and the pan compiler will, by default, generate an XML profile for each processed object template. An object template may contain any of the pan statements. Statements that operate on paths may contain only absolute paths.

Object template names may be namespaced, allowing organization of object templates in directory structures as is done for other templates. For the automatic loading mechanism to find object templates, the root directory containing them must be specified explicitly in the load path (either on the command line or via the `LOADPATH` variable).

2.2. Ordinary Templates

An ordinary template uses no template modifier in the declaration. These templates may contain any pan statement, but statements must operate only on absolute paths.

2.3. Declaration Templates

A template declared with a `declaration` modifier is a declaration template. These template may contain only those pan statements that do not modify the machine profile. That is, they may contain only **type**, **bind**, **variable**, and **function** statements. A declaration template will only be executed once for each processed object template no matter how many times it is included. It will be executed when the first include statement referencing the template is encountered.

2.4. Unique Templates

A template defined with the `unique` modifier behaves like an ordinary template except that it will only be included once for each processed object template. It has the same restrictions as an ordinary template. It will be executed when the first include statement referencing the template is encountered.

2.5. Structure Templates

A template declared with the `structure` modifier may only contain **include** statements and assignment statements that operate on relative paths. The **include** statements may only reference other structure templates. Structure templates are an alternative for creating nlists and are used via the `create` function.

3. Statements

3.1. assignment

Assignment statements are used to modify a part of the configuration tree by replacing the subtree identified by its path by the result of the execution a DML block. This result can be a single property or a resource holding any number of elements. There are conditional and unconditional assignments:

```
[ final ] path ?= dml;
[ final ] path = dml;
```

where the path is represented by a string literal. Single-quoted strings are slightly more efficient, but double-quoted strings work as well. The conditional form (?=) will only execute the DML block and assign a value if the named path does not exist or contains the `undef` value.

The assignment will create parents of the value that do not already exist.

If a value already exists, the pan compiler will verify that the new value has a compatible type. If not, it will terminate the processing with an error.

If the `final` modifier is used, then the path and any children of that path may not be subsequently modified. Attempts to do so will result in a fatal error.

3.2. include

The **include** statement acts as if the contents of the named template were included literally at the point the include statement is executed. Two variants of the **include** statement are permitted:

```
include template-name;
include {dml};
```

The first variant includes the named template; the template is written as a token without quotes. The second variant executes the given DML block. The block must evaluate to a string, `undef`, or `null`. If the result is `undef` or `null`, the **include** statement does nothing; if the result is a string, the named template is loaded and executed. Any other type will generate an error. The braces in the second variant are required.

Ordinary templates may be included multiple times, but loops are not permitted. Templates marked as `declaration` or `unique` templates will be only included once where first encountered.

There are some restrictions on what types of templates can be included. Object templates cannot be included. Structure templates can only include and be included by other structure templates. Declaration templates can only include other declaration templates. All other situations are allowed.

3.3. function

Functions can be defined by the user. These are arbitrary DML blocks bound to an identifier. Once defined, functions can be called from any subsequent DML block. Functions may only be defined once; attempts to redefine an existing function will cause the compilation to abort. The function definition syntax is:

```
function identifier = dml;
```

See the Function section for more information on user-defined functions and a list of built-in functions.

Note that the compiler keeps distinct function and type namespaces. One can define a function and type with the same names.

3.4. type

Type definitions are critical for the validation of the generated machine profiles. Types can be built up from the primitive pan types and arbitrary validation functions. New types can be defined with

```
type identifier = type-spec;
```

A type may be defined only once; attempts to redefine an existing type will cause the compilation to abort. Types referenced in the type-spec must already be defined. See the Type section for more details on the syntax of the type specification.

Note that the compiler keeps distinct function and type namespaces. One can define a function and type with the same names.

3.5. variable

Global variables can be defined via a **variable** statement. These may be referenced from any DML block after being defined. They may not be modified from a DML block; they can only be modified from a **variable** statement. Like the assignment statement there are conditional and unconditional forms:

```
[ final ] variable identifier ?= dml;
[ final ] variable identifier = dml;
```

For the conditional form, the DML block will only be evaluated and the assignment done if the variable does not exist or has the `undef` value.

If the `final` modifier is used, then the variable may not be subsequently modified. Attempts to do so will result in a fatal error.

Pan provides several automatic global variables: `OBJECT`, `SELF`, `FUNCTION`, and `LOADPATH` (and their deprecated lowercase equivalents). `OBJECT` contains the name of the object template being evaluated; it is a final variable. `SELF` is the current value of a path referred to in an assignment or variable statement. The `SELF` reference cannot be modified, but children of `SELF` may be. `FUNCTION` contains the name of the current function, if it exists. `FUNCTION` is a final variable. `LOADPATH` can be used to modify the load path used to locate template for the **include** statement.

Any valid identifier may be used to name a global variable. To avoid conflicts with local variables, however, global variables are usually named in all uppercase; local variables in all lowercase.

3.6. bind

The **bind** statement binds a type definition to a path. Multiple types may be bound to a single path. During the validation phase, the value corresponding to the named path will be checked against the bound types. There are two variants:

```
bind path = type-spec;
type path = type-spec;
```

See the Type section for a complete description of the *`type-spec`* syntax. Note that the second form is deprecated and will disappear in a future release.

3.7. valid

The **valid** statement binds a validation DML block to a path. It has the form:

```
valid path = DML;
```

This is a convenience statement and has exactly the same effect as the statement:

```
bind path = element with DML;
```

The pan compiler internally implements this statement as the **bind** statement above.

4. Types

The following statement will bind an existing type definition (either a built-in definition or a user-defined one) to a path in a machine configuration:

```
bind path = type-spec;
```

where 'path' is a valid path name and *`type-spec`* is either a type specification or name of an existing type. The syntax:

```
type path = type-spec; # DEPRECATED
```

will also work, but its use is deprecated. Support for the latter form will disappear in a future release of the compiler.

Full type specifications are of the form:

```
identifier = constant with validation-dml
```

where 'constant' is a DML block that evaluates to a compile-time constant (the default value), and the 'validation-dml' is a DML block that will be run to validate paths associated with this type. Both the default value and validation block are optional. The identifier can be any legal name with an optional array specifier and/or range afterwards. For example, an array of 5 elements is written int[5] or a string of length 5 to 10 characters string(5..10).

4.1. Primitive Types

There are five primitive, atomic types in the pan language:

- boolean
- long
- double
- string
- link

The "link" type appears as a string, but must be a valid path name that exists at validation time. In addition, there are two primitive collection types:

- list
- nlist

The 'list' is an ordered list of elements. The named list (nlist) associates a string key with a value; these are also known as hashes or associative lists. These seven comprise the primitive types in the pan language.

4.2. User-Defined Types

Users can create new types built up from the primitive types and with optional validation functions. The general format for creating a new type is:

```
type identifier = type-spec;
```

where the general form for a type specification 'type-spec' is given above.

Probably the easiest way to understand the type definitions is by example. The following are "alias" types that associate a new name with an existing type, plus some restrictions.

```
type ulong1 = long with self >= 0;
type ulong2 = long(0..);
type port = long(0..65535);
type short_string = string(..255);
type small_even = long(-16..16) with self %2 == 0;
```

Similarly one can create link types for elements in the machine configuration:

```
type mylink = long(0..)* with match(self, 'r$');
```

Values associated to this type must be a string ending with 'r'; the value must be a valid path that references an unsigned long value.

Slightly more complex is to create uniform collections:

```
type long_list = long[10];
type matrix = long[3][4];
type double_nlist = double{};
```

```
type small_even_nlist = small_even{};
```

Here all of the elements of the collection have the same type. The last example shows that previously-defined, user types can be used as easily as the built-in primitive types.

A record is an nlist that explicitly names and types its children. A record is by far, the most used type definition. For example, the type definition:

```
type cpu = {
  'vendor' : string
  'model' : string
  'speed' : double
  'fpu' ? boolean
};
```

defines an nlist with four children named 'vendor', 'model', etc. The first three use a colon (":") in the definition and are consequently required; the last uses a question mark ("?") and is optional. As defined, no other children may appear in nlists of this type. However, one can make the record extensible with:

```
type cpu = extensible {
  'vendor' : string
  'model' : string
  'speed' : double
  'fpu' ? boolean
};
```

This will check the types of 'vendor', 'model', etc., but will also allow children of the nlist with different unlisted names to appear. This provides some limited subclassing support. Each of the types for the children can be a full type specification and may contain default values and/or validation blocks. One can also attach default values or validation blocks to the record as a whole.

5. Functions

Within a Data Manipulation Language (DML) block, user-defined and built-in functions may be called. The pan language uses a typical syntax for calling a function:

```
identifier(arg1, arg2, ...);
```

where 'identifier' is a valid pan identifier and the arguments are passed to the function as a comma-separated list. The arguments may be expressions, in which case those expressions will be completely evaluated before calling the function.

All functions return a value, although it might be `undef`.

5.1. User-Defined Functions

The pan language permits user-defined functions. These functions are essentially a DML block bound to an identifier. Only one DML block may be assigned to a given identifier. Attempts to redefine an existing function will cause the execution to be aborted. The syntax for defining a function is:

```
function identifier = DML;
```

where identifier is a valid pan identifier and DML is the block to bind to this identifier.

When the function is called, the DML will have the variables `ARGC` and `ARGV` defined. (The deprecated names `argc` and `argv` will also be defined.) The variable `ARGC` contains the number of arguments passed to the function; `ARGV` is a list containing the values of the arguments.

Note that `ARGV` is a standard pan list. Consequently, passing null values (intended to delete elements) to functions can have non-obvious effects. For example, the call:

```
f(null);
```

will result is an empty ARGV list because the null value deletes the nonexistent element ARGV[0].

The pan language does no automatic checking of the number or types of arguments. The DML block that defines the function must make all of these checks explicitly and use the error() function to emit an informative message in case of an error.

Recursive calls to a function are permitted. However, the call depth is limited (by an option when the compiler is invoked) to avoid infinite recursion. Typically, the maximum is a small number like 10.

The following example defines a function that checks if the number of arguments is even and are all numbers:

```
function even_numbers = {

  if (ARGC%2 != 0) {
    error('number of arguments must be even');
  };

  foreach (k, v, ARGV) {
    if (! is_number(v)) {
      error('non-numeric argument found');
    };
  };

};
```

5.2. Built-In Functions

Table 1. String Manipulation Functions

format(3)	Generate a formatted string based on the formatting parameters and the values provided.
index(3)	Return the index of a substring or -1 if the substring is not found.
length(3)	Gives the length of a string.
match(3)	Return a boolean indicating if a string matches the given regular expression.
matches(3)	Return an array containing the matched string and matched groups for a given string and regular expression.
replace(3)	Replace all occurrences of a substring within a given string.
splice(3)	Remove a substring and optionally replace it with another.
split(3)	Split a string based on a given regular expression and return an array of the results.
substr(3)	Extract a substring from the given string.
to_lowercase(3)	Change all of the characters in a string to lowercase (using the US locale).
to_uppercase(3)	Change all of the characters in a string to uppercase (using the US locale).

Table 2. Debugging Functions

debug(3)	Print a debugging message to the standard error stream. Returns the message or `undef`.
error(3)	Print an error message to the standard error and terminate processing.
traceback(3)	Print an error message to the standard error along with a traceback. Returns `undef`.
deprecated(3)	Print a warning message to the standard error if required by the deprecation level in effect. Returns `the message or undef`.

Table 3. Encoding and Decoding Functions

base64_decode(3)	Decode a string that is encoded using the Base64 standard.
base64_encode(3)	Encode a string using the Base64 standard.
escape(3)	Escape characters within the string to ensure string is a valid nlist key (path term).
unescape(3)	Transform an escaped string into its original form.

Table 4. Resource Manipulation Functions

append(3)	Add a value to the end of a list.
create(3)	Create an nlist from the named structure template.
first(3)	Initialize an iterator over a resource. Returns a boolean to indicate if more values exist in the resource.
nlist(3)	Create an nlist from the given key/value pairs given as arguments.
key(3)	Find the n'th key in an nlist.
length(3)	Get the number of elements in the given resource.
list(3)	Create a list from the given arguments.
merge(3)	Perge two resources into a single one. This function always creates a new resource and leaves the arguments untouched.
next(3)	Extract the next value while iterating over a resource. Returns a boolean to indicate if more values exist in the resource.
prepend(3)	Add a value to the beginning of a list.
splice(3)	Remove a section of a list and optionally replace removed values with those in a given list.

Table 5. Type Checking Functions

is_boolean(3)	Check if the argument is a boolean value. If the argument is a simple variable reference and the referenced variable does not exist, the function will return false rather than raising an error.
is_defined(3)	Check if the argument is a value other than `null` or `undef`. If the argument is a simple variable

	reference and the referenced variable does not exist, the function will return false rather than raising an error.
is_double(3)	Check if the argument is a double value. If the argument is a simple variable reference and the referenced variable does not exist, the function will return false rather than raising an error.
is_list(3)	Check if the argument is a list. If the argument is a simple variable reference and the referenced variable does not exist, the function will return false rather than raising an error.
is_long(3)	Check if the argument is a long value. If the argument is a simple variable reference and the referenced variable does not exist, the function will return false rather than raising an error.
is_nlist(3)	Check if the argument is an nlist. If the argument is a simple variable reference and the referenced variable does not exist, the function will return false rather than raising an error.
is_null(3)	Check if the argument is a `null`. If the argument is a simple variable reference and the referenced variable does not exist, the function will return false rather than raising an error.
is_number(3)	Check if the argument is either a long or double value. If the argument is a simple variable reference and the referenced variable does not exist, the function will return false rather than raising an error.
is_property(3)	Check if the argument is a property (long, double, or string). If the argument is a simple variable reference and the referenced variable does not exist, the function will return false rather than raising an error.
is_resource(3)	Check if the argument is a list or nlist. If the argument is a simple variable reference and the referenced variable does not exist, the function will return false rather than raising an error.
is_string(3)	Check if the argument is a string value. If the argument is a simple variable reference and the referenced variable does not exist, the function will return false rather than raising an error.

Table 6. Type Conversion Functions

to_boolean(3)	Convert the argument to a boolean. Any number other than 0 and 0.0 is `true`. The empty string and the string 'false' (ignoring case) return `false`. Any other string will return `true`. If the argument is a resource, an error will occur.
to_double(3)	Convert the argument to a double value. Strings will be parsed to create a double value; any literal form of a double is valid. Boolean values will convert to `0.0` and `1.0` for `false` and `true`, respectively. Long values are converted to the cor-

	responding double value. Double values are unchanged.
to_long(3)	Convert the argument to a long value. Strings will be parsed to create a long value; any literal form of a long is valid (e.g. hex or octal literals). Boolean values will convert to `0` and `1` for `false` and `true`, respectively. Double values are rounded to the nearest long value. Long values are unchanged.
to_string(3)	Convert the argument to a string. The function will return a string representation for any argument, including list and nlist.

Table 7. Miscellaneous Functions

clone(3)	Create a deep copy of the given value.
delete(3)	Delete a local variable or child of a local variable.
exists(3)	Return true if the given argument exists. The argument can either be a variable reference, path, or template name.
path_exists(3)	Return true if the given path exists. The argument must be an absolute or external path.
if_exists(3)	For a given template name, return the template name if it exists or undef if it does not. This can be used with the include statement for a conditional include.
return(3)	Interrupt the normal flow of processing and return the given value as the result of the current frame (either a function call or the main DML block).
value(3)	Retrieve the value associated with the given path. The path may either be an absolute or external path.

6. Data Manipulation Language

Many of the top-level pan language statements take a Data Manipulation Language (DML) block. Although the DML is not intended as a complete high-level programming language, it does provide many features of such a language and makes the pan language more flexible than a pure declarative language.

Overall the DML looks very similar to stripped-down c or java. The syntax and operators will be familiar to programmers of either language. The principal difference is that all DML statements return a value, like Lisp. The value of a block of statements is the value of the last statement executed.

6.1. Literals

6.1.1. Boolean Literals

There are exactly two possible boolean values: `true` and `false`. They must appear as an unquoted word and completely in lowercase.

6.1.2. Long Literals

Long literals may be given in decimal, hexadecimal, or octal format. A decimal literal is a sequence of digits starting with a number other than zero. A hexadecimal literal starts with the '0x' or '0X' and

is followed by a sequence of hexadecimal digits. An octal literal starts with a zero is followed by a sequence of octal digits. Examples:

```
123  # decimal long literal
0755 # octal long literal
0xFF # hexadecimal long literal
```

Long literals are represented internally as an 8-byte signed number. Long values that cannot be represented in 8 bytes will cause a syntax error to be thrown.

6.1.3. Double Literals

Double literals represent a floating point number. A double literal must start with a digit and must contain either a decimal point or an exponent. Examples:

```
0.01
3.14159
1e-8
1.3E10
```

Note that '.2' is not a valid double literal; this value must be written as '0.2'.

Double literals are represented internally as an 8-byte value. Double values that cannot be represented in 8 bytes will cause a syntax error to be thrown.

6.1.4. String Literals

The string literals can be expressed in three different forms. They can be of any length and can contain any character, including the NULL byte.

Single quoted strings are used to represent short and simple strings. They cannot span several lines and all the characters will appear verbatim in the string, except the doubled single quote which is used to represent a single quote inside the string. For instance:

```
'foo'
'it''s a sentence'
'^\d+\.\d+$'
```

This is the most efficient string representation and should be used when possible.

Double quoted strings are more flexible and use the backslash to represent escape sequences. For instance:

```
"foo"
"it's a sentence"
"Java-style escapes: \t (tab) \r (carriage return) \n (newline)"
"Hexadecimal escapes: \x3d (=) \x00 (NULL byte) \x0A (newline)"
"Miscellaneous escapes: \" (double quote) \\ (backslash)"
"this string spans two lines and\
does not contain a newline"
```

Invalid escape sequences will cause a syntax error to be thrown.

Multi-line strings can be represented using the 'here-doc' syntax, like in shell or Perl.

```
"test" = "foo" + <<EOT + "bar";
this code will assign to the path "test" the string
made of 'foo', plus this text including the final newline,
plus 'bar'...
EOT
```

The contents of the 'here-doc' are treated as a single-quoted string. That is, no escape processing is done.

The easiest solution to put binary data inside pan code is to base64 encode it and put it inside ”here-doc” strings like in the following example:

```
"/system/binary/stuff" = base64_decode(<<EOT);
H4sIAOwLyDwAA02PQQ7DMAgE731FX9BT1f8QZ52iYhthEiW/r2SitCdmxCK0E3W8no+36n2G
8UbOrYYWGROCgurBe4JeCexI2ahgWF5rulaLtImkDxbucS0tcc3t5GXMAqeZnIYo+TvAmsL8
GGLobbUUX7pT+pxkXJc/5Bx5p0ki7Cgq5KccGrCR8PzruUfP2xfJgVqHCgEAAA==
EOT
```

The `base64_decode()` function is one of the built-in pan functions.

6.1.4.1. Path Literals

Pan paths are represented as string literals; any of the standard forms for a string literal can be used to represent a path. There are three different types of paths: external, absolute, and relative. An external path explicitly references an object template. There are two forms that are currently supported:

```
my/external/object:/some/absolute/path
myobject:/some/absolute/path
```

The first form allows namespace object templates to be referenced; the second form does not. The second form is deprecated.

An absolute path starts at the top of a configuration tree and identifies a node within the tree. All absolute paths start with a slash ("/") and are followed by a series of terms that identify a specific child of each resource. A bare slash ("/") refers to the full configuration tree. The allowed syntax for each term in the path is described below.

A relative path refers to a path relative to a structure template. Relative paths do not start with a slash, but otherwise are identical to the absolute paths.

Terms may consist of letters, digits, underscores, hyphens, and pluses. Terms beginning with a digit must be a valid long literal. Terms that contain other characters must be escaped, either by using the escape() function within a DML block or by enclosing the term within braces for a path literal. For example, the following creates an absolute path with three terms:

```
/alpha/{a/b}/gamma
```

The second term is equivalent to escape("a/b").

6.1.4.2. Regular Expressions

Regular expressions are written as a standard pan string literals. The regular expression syntax is compatible with the Perl regular expression syntax. (The actual implementation exposes the java regular expression syntax, which is largely compatible with the Perl syntax.) Because certain characters have a special meaning in pan double quoted strings, characters like backslashes will need to be escaped. It is preferable to use single-quoted strings for regular expression literals.

6.1.5. Special Literals

undef

The undef literal can be used to represent the undefined element, i.e. an element which is neither a property nor a resource.

The undefined element cannot be written to a final machine profile and most built-in functions will report a fatal error when processing it. It can be used to mark an element that must be overwritten during the processing.

null

The null value deletes the path or global variable to which it is assigned. Most operations and functions will report an error if this value is processed directly.

6.2. Variables

To ease data handling, you can use variables in any DML expression. They are by default lexically scoped to the *outermost* enclosing DML expression. They do not need to be declared before they are used. The local variables are destroyed once the outermost enclosing DML block terminates.

As a first approximation, variables work the way you expect them to work. They can contain properties and resources and you can easily access resource children using square brackets:

```
# populate /table which is a nlist
'/table/red' = 'rouge';
'/table/green' = 'vert';

'/test' = {
  x = list('a', 'b', 'c'); # x is a list
  y = value('/table');     # y is a nlist
  z = x[1] + y['red'];     # z is a string
  return(length(z));       # this will be 6
};
```

Local variables are subject to primitive type checking. So the primitive type of a local variable cannot be changed unless the variable is assigned a value of 'undef' or 'null' between the type-changing assignments.

Global variables (defined with the 'variable' statement) can be read from the DML block. Global variables may not be modified from within the block; attempting to do so will abort the execution.

Global and local variables share the same namespace. Consequently, there may be unintended naming conflicts between them. The best practice to avoid this, is to name all local variables with all lowercase letters and all global variables with all uppercase letters.

6.3. Operators

The operators available in the pan Data Manipulation Language (DML) are very similar to those in the java or c languages. The following tables summarize the DML operators. The valid primitive types for each operator are indicated. Those marked with "number" will take either long or double arguments. In the case of binary operators, the result will be promoted to a double if the operands are mixed.

Table 8. Unary DML Operators

+	number	preserves sign of argument
-	number	changes sign of argument
~	long	bitwise not
!	boolean	logical not

Table 9. Binary DML Operators

+	number	addition
+	string	string concatenation
-	number	subtraction
*	number	multiplication

/	number	division
%	long	modulus
&	long	bitwise and
|	long	bitwise or
^	long	bitwise exclusive or
&&	boolean	logical and (short-circuit logic)
||	boolean	logical or (short-circuit logic)
==	number	equal
==	string	lexical equal
!=	number	not equal
!=	string	lexical not equal
>	number	greater than
>	string	lexical greater than
>=	number	greater than or equal
>=	string	lexical greater than or equal
<	number	less than
<	string	lexical less than
<=	number	less than or equal
<=	string	lexical less than or equal

Table 10. Operator Precedence (lowest to highest)

||
&&
|
^
&
==, !=
<, <=, >, >=
+ (binary), - (binary)
*, /, %
+ (unary), - (unary), !, ~

6.4. Flow Control

DML contains four statements that permit non-linear execution of code within a DML block. The `if` statement allows conditional branches, the `while` statement allows looping over a DML block, the `for` statement allows the same, and the `foreach` statement allows iteration over an entire resource (list or nlist).

These statements, like all DML statements, return a value. Be careful of this, because unexecuted blocks generally will return 'undef', which may lead to non-intuitive behavior.

6.4.1. Branching (`if` statement)

The `if` statement allows conditional execution of a DML block. The statement may include an `else` clause that will be executed if the condition is `false`. The syntax is:

```
if ( condition-dml ) true-dml;
if ( condition-dml ) true-dml else false-dml;
```

where all of the blocks may either be a single DML statement or a true DML block.

The value returned by this statement is the value returned by the *true-dml* or *false-dml* block, whichever is actually executed. If the `else` clause is not present and the *condition-dml* is false, the if statement returns `undef`.

6.4.2. Looping (`while` and `for` statements)

Simple looping behavior is provided by the `while` statement. The syntax is:

```
while ( condition-dml ) body-dml;
```

The loop will continue until the *condition-dml* evaluates as `false`. The value of this statement is that returned by the *body-dml* block. If the *body-dml* block is never executed, then `undef` is returned.

The pan language also contains a for statement that in many cases provides a more concise syntax for many types of loops. The syntax is:

```
for (initialization-dml; condition-dml; increment-dml) body-dml;
```

The *initialization-dml* block will first be executed. Before each iteration the *condition-dml* block will be executed; the *body-dml* will only be executed (again) if the condition evaluates to `true`. After each iteration, the *increment-dml* block is executed. If the condition never evaluates to `true`, then the value of the statement will be that of the *initialization-dml*. All of the DML blocks must be present, but those not of interest can be defined as just `undef`.

Note that the compiler usually enforces an iteration limit to avoid infinite loops. Loops exceeding the iteration limit will cause the compiler to abort the execution.

6.4.3. Iteration (`foreach` statement)

The 'foreach' statement allows iteration over all of the elements of a list or nlist. The syntax is:

```
foreach (key; value; resource) body-dml;
```

This will cause the body-dml to be executed once for each element in resource (a list or nlist). The local variables 'key' and 'value' (you can choose these names) will be set at each iteration to the key and value of the element. For a list, the key is the element's index. The iteration will always occur in the natural order of the resource: ordinal order for lists and lexical order of the keys for nlists.

The value returned will be that of the last iteration of the body-dml. If the body-dml is never executed (for an empty list or nlist), 'undef' will be returned.

The 'foreach' statement is not subject to the compiler's iteration limit. By definition, the resource has a finite number of entries, so this safeguard is not needed.

This form of iteration should be used in preference to the first/next functions or the key function whenever possible. It is more efficient than the functional forms and less prone to error.

7. Other Features

7.1. Annotations

An annotation mechanism has been added to the pan language to allow a set of key-value pairs to be associated with various language-level elements. This was originally intended to help with the

generation of documentation, but the general nature of the solution may lend itself to other uses. The basic syntax of an annotation is:

```
@(content) | @ [content] | @{content}
```

where *content* is either a set of key-value pairs or a block of text. The chosen delimiter (parenthesis, bracket, or brace) cannot appear in the enclosed *content*. Annotations may appear before any token in the pan language and are logically attached to the following token. The compiler will process annotations when building machine configurations, but ignore the contents. Syntactically invalid annotations will cause the compilation to fail.

A set of key-value pairs inside of an annotation are written like:

```
@(
  key1 = value1
  key2 = value2
)
```

The key can be any sequence of letters, digits, and the characters "_.+-/". The value associated with the key is the string starting with the first non-whitespace character after the equals sign up to, but not including, the end of line or terminating delimiter of the annotation. The syntax is tolerant of whitespace with whitespace before the key and around the equals sign being ignored. Lines consisting of only whitespace are also ignored.

Long values may be split between lines by terminating intermediate lines with a single backslash. For example,

```
@(
  key1 = some \
long \
value
)
```

Will assign the value "some long value" to the key "key1". Note that the newlines after the backslashes are *not* included in the value.

Multi-line values may be defined by quoting the value, such as the following:

```
@(
  key1 = "some
long
value"
)
```

The value consists of the three words "some", "long", and "value" separated by newlines. Either single or double quotes may be used, but the value itself may not contain the chosen delimiter. The quote must be the first non-whitespace character after the equals sign to start a multi-line value.

An abbreviated format of an annotation allows for a single descriptive value. For example,

```
@(A long descriptive text
that can span multiple lines)
```

In this case, the value is treated as a multi-line value and is assigned to the key "desc". That is, the above is equivalent to:

```
@( desc = "A long descriptive text
that can span multiple lines" )
```

The only limitation with this abbreviated syntax is that the equals sign may not appear in the first, non-blank line of the value.

It is expected that pan compilers will provide a mechanism to recover the annotation information to allow the automatic processing of those annotations. See the compiler manual to understand if and how that can be done.

A. Built-In Function Reference

Name

append — adds a value to the end of a list

Synopsis

```
list append(value);

element value;

list append(target, value);

list target;
element value;

list append(target, value);

variable_reference target;
element value;
```

Description

The append function will add the given value to the end of the target list. There are three variants of this function. For all of the variants, an explicit null value is illegal and will terminate the compilation with an error.

The first variant takes a single argument and always operates on SELF. It will directly modify the value of SELF and give the modified list (SELF) as the return value. If SELF does not exist, is undef, or is null, then an empty list will be created and the given value appended to that list. If SELF exists but is not a list, an error will terminate the compilation. This variant cannot be used to create a compile-time constant.

```
# /result will have the values 1 and 2 in that order
'/result' = list(1);
'/result' = append(2);
```

The second variant takes two arguments. The first argument is a list value, either a literal list value or a list calculated from a DML block. This version will create a copy of the given list and append the given value to the copy. The modified copy is returned. If the target is not a list, then an error will terminate the compilation. This variant can be used to create a compile-time constant as long as the target expression does not reference information outside of the DML block by using, for example, the value function.

```
# /result will have the values 1 and 2 in that order
# /x will only have the value 1
'/x' = list(1);
'/result' = append(value('/x'), 2);
```

The third variant also takes two arguments, where the first value is a variable reference. This variant will take precedence over the second variant. This variant will directly modify the referenced variable and return the modified list. If the referenced variable does not exist, it will be created. As for the other forms, if the referenced target exists and is not a list, then an error will terminate the compilation. SELF or descendants of SELF can be used as the target. This variant can be used to create a compile-time constant if the referenced variable is an *existing* local variable. Referencing a global variable (except via SELF) is not permitted as modifying global variables from within a DML block is forbidden.

```
# /result will have the values 1 and 2 in that order
'/result' = {
  append(x, 1); # will create local variable x
```

```
  append(x, 2);
};
```

Name

base64_decode — decodes a string that has been encoded in base64 format

Synopsis

```
string base64_decode(encoded);

string encoded;
```

Description

The `base64_decode` function will return the unencoded value of the base64 (RFC 2045) encoded argument. If the argument is not a valid base64 encoded value a fatal error will occur.

```
# /result have the string value 'hello world'
'/result' = base64_decode('aGVsbG8gd29ybGQ=');
```

Name

base64_encode — encodes a string in base64 format

Synopsis

```
string base64_encode(unencoded);

string unencoded;
```

Description

The `base64_encode` function will return the base64 (RFC 2045) encoded format of the argument.

```
# /result have the string value 'aGVsbG8gd29ybGQ='
'/result' = base64_encode('hello world');
```

Name

clone — returns a clone (copy) of the argument

Synopsis

```
element clone(arg);

element arg;
```

Description

The `clone` function may return a clone (copy) of the argument. If the argument is a resource, the result will be a "deep" copy of the argument; subsequent changes to the argument will not affect the clone and vice versa. Because properties are immutable internally, this function will not actually copy a property instead returning the argument itself.

Name

create — create an nlist from a structure template

Synopsis

```
nlist create(tpl_name, );

string tpl_name;
...;
```

Description

The `create` function will return an nlist from the named structure template. The optional additional arguments are key, value pairs that will be added to the returned nlist, perhaps overwriting values from the structure template. The keys must be strings that contain valid nlist keys. The values can be any element. Null values will delete the given key from the resulting nlist.

```
# description of CD mount entry with the device undefined
# (in file 'mount_cdrom.tpl')
structure template mount_cdrom;
'device' = undef;
'path' = '/mnt/cdrom';
'type' = 'iso9660';
'options' = list('noauto', 'owner', 'ro');

# use from within another template
'/system/mounts/0' = create('mount_cdrom', 'device', 'hdc');

# the above is equivalent to the following two lines
'/system/mounts/0' = create('mount_cdrom');
'/system/mounts/0/device' = 'hdc';
```

Name

debug — print debugging information to the console

Synopsis

```
string debug(msg);

string msg;
```

Description

This function will print the given string to the console (on stdout) and return the message as the result. This functionality must be activated either from the command line or via a compiler option (see compiler manual for details). If this is not activated, the function will not evaluate the argument and will return undef.

Name

delete — delete the element identified by the variable expression

Synopsis

```
undef delete(arg);

variable_expression arg;
```

Description

This function will delete the element identified by the variable expression given in the argument and return undef. The variable expression can be a simple or subscripted variable reference (e.g. x, x[0], x['abc'][1], etc.). Only variables local to a DML block can be modified with this function. Attempts to modify a global variable will cause a fatal error. For subscripted variable references, this function has the same effect as assigning the variable reference to null.

```
# /result will contain the list ('a', 'c')
'/result' = {
  x = list('a', 'b', 'c');
  delete(x[1]);
  x;
};
```

Name

deprecated — print deprecation warning to console

Synopsis

```
string deprecated( level, msg);

long level;
string msg;
```

Description

This function will print the given string to the console (on stderr) and return the message as the result, if *level* is less than or equal to the deprecation level given as a compiler option. If the message is not printed, the function returns undef. The value of *level* must be non-negative.

Name

error — print message to console and abort compilation

Synopsis

```
void error(msg);

string msg;
```

Description

This function prints the given message to the console (stderr) and aborts the compilation. This function cannot appear neither in variable subscripts nor in function arguments; a fatal error will occur if found in either place.

```
# a user-defined function requiring one argument
function foo = {

  if (ARGC != 1) {
    error("foo(): wrong number of arguments: " + to_string(ARGC));
  };

  # normal processing...
};
```

Name

escape — escape non-alphanumeric characters to allow use as nlist key

Synopsis

```
string escape(str);

string str;
```

Description

This function escapes non-alphanumeric characters in the argument so that it can be used inside paths, for instance as an nlist key. Non-alphanumeric characters are replaced by an underscore followed by the hex value of the character. If the string begins with a digit, the initial digit is also escaped. If the argument is the empty string, the returned value is a single underscore '_'.

```
# /result will have the value '1_2b1'
'/result' = escape('1+1');
```

Name

exists — determines if a variable expression, path, or template exists

Synopsis

```
boolean exists(var);

variable_expression var;

boolean exists(path);

string path;

boolean exists(tpl);

string tpl;
```

Description

This function will return a boolean indicating whether a variable expression, path, or template exists. If the argument is a variable expression (with or without subscripts) then this function will return true if the given variable exists; the value of referenced variable is not used. If the argument is not a variable reference, the argument is evaluated; the value must be a string. If the resulting string is a valid external or absolute path, the path is checked. Otherwise, the string is interpreted as a template name and the existence of this template is checked.

Note that if the argument is a variable expression, only the existence of the variable is checked. For example, the following code will always leave `r` with a value of `true`.

```
v = '/some/absolute/path';
r = exists(v);
```

If you want to test the path, remove the ambiguity by using a construct like the following:

```
v = '/some/absolute/path';
r = exists(v+'');
```

The value of `r` in this case will be `true` if `/some/absolute/path` exists or `false` otherwise.

Name

first — initialize an iterator over a resource and return first entry

Synopsis

```
boolean first(r, key, value);

resource r;
variable_expression key;
variable_expression value;
```

Description

This function resets the iterator associated with `r` so that it points to the beginning of the resource. It will return `false` if the resource is empty; `true`, otherwise. If the resource is not empty, then it will also set the variable identified by `key` to the child's index and the variable identified by `value` to the child's value. Either `key` or `value` may be `undef`, in which case no assignment is made. For a list resource `key` is the child's numeric index; for an nlist resource, the string value of the key itself. An example of using `first` with a list:

```
# compute the sum of the elements inside numlist
numlist = list(1, 2, 4, 8);
sum = 0;
ok = first(numlist, k, v);
while (ok) {
  sum = sum + v;
  ok = next(numlist, k, v);
};
# value of sum will be 15
```

An example of using `first` with an nlist:

```
# put the list of all the keys of table inside keys
table = nlist("a", 1, "b", 2, "c", 3);
keys = list();
ok = first(table, k, v);
  while (ok) {
  keys[length(keys)] = k;
  ok = next(table, k, v);
};
# keys will be ("a", "b", "c")
```

Name

format — format a string by replacing references to parameters

Synopsis

```
string format(fmt, param, );

string fmt;
property param;
...;
```

Description

The `format` function will replace all references within the *`fmt`* string with the values of the referenced properties. This provides functionality similar to the c-language's `printf` function. The syntax of the *`fmt`* string follows that provided in the java language; see the `Formatter` entry for full details.

Name

if_exists — check if a template exists, returning template name if it does

Synopsis

```
string|undef if_exists(tpl);

string tpl;
```

Description

The `if_exists` function checks if the named template exists on the current load path. If it does, the function returns the name of the template. If it does not, `undef` is returned. This can be used to conditionally include a template:

```
include {if_exists('my/conditional/template')};
```

This function should be used with caution as this brings in dependencies based on the state of the file system and may cause dependency checking to be inaccurate.

Name

index — finds substring within a string or element within a resource

Synopsis

```
long index(sub, arg, start);

string sub;
string arg;
long start;

long index(sub, list, start);

property sub;
string list;
long start;

string index(sub, arg, start);

property sub;
nlist arg;
long start;

long index(sub, arg, start);

nlist sub;
list arg;
long start;

string index(sub, arg, start);

nlist sub;
nlist arg;
long start;
```

Description

The `index` function returns the location of a substring within a string or an element within a resource. In detail the five different forms perform the following actions.

The first form searches for the given substring inside the given string and returns its position from the beginning of the string or `-1` if not found; if the third argument is given, starts initially from that position.

```
'/s1' = index('foo', 'abcfoodefoobar'); # 3
'/s2' = index('f0o', 'abcfoodefoobar'); # -1
'/s3' = index('foo', 'abcfoodefoobar', 4); # 8
```

The second form searches for the given property inside the given list of properties and returns its position or `-1` if not found; if the third argument is given, starts initially from that position; it is an error if `sub` and `arg`'s children are not of the same type.

```
# search in a list of strings (result = 2)
"/l1" = index("foo", list("Foo", "FOO", "foo", "bar"));

# search in a list of longs (result = 3)
"/l2" = index(1, list(3, 1, 4, 1, 6), 2);
```

The third form searches for the given property inside the given named list of properties and returns its name or the empty string if not found; if the third argument is given, skips that many matching children; it is an error if `sub` and `arg`'s children are not of the same type.

```
# simple color table
'/table' = nlist('red', 0xf00, 'green', 0x0f0, 'blue', 0x00f);

# result will be the string 'green'
'/name1' = index(0x0f0, value('/table'));

# result will be the empty string
'/name2' = index(0x0f0, value('/table'), 1);
```

The fourth form searches for the given nlist inside the given list of nlists and returns its position or `-1` if not found. The comparison is done by comparing all the children of `sub`, these children must all be properties. If the third argument is given, starts initially from that position. It is an error if `sub` and `arg`'s children are not of the same type or if their common children don't have the same type.

```
# search a record in a list of records (result = 1, the second nlist)
'/l1' = index(
             nlist('key', 'foo'),
             list(
                  nlist('key', 'bar', 'val', 101),
                  nlist('key', 'foo', 'val', 102)
                 )
            );
```

The last form searches for the given nlist inside the given nlist of nlists and returns its name or the empty string if not found. If the third argument is given, the function skips that many matching children. It is an error if `sub` and `arg`'s children are not of the same type or if their common children don't have the same type.

Name

is_boolean — checks to see if the argument is a double

Synopsis

```
boolean is_boolean(arg);

element arg;
```

Description

The `is_boolean` function will return `true` if the argument is a boolean value; it will return `false` otherwise.

Name

is_defined — checks to see if the argument is anything but `undef` or `null`

Synopsis

```
boolean is_defined(arg);

element arg;
```

Description

The `is_defined` function will return a `true` value if the argument is anything but `undef` or `null`; it will return `false` otherwise.

Name

is_double — checks to see if the argument is a double

Synopsis

```
boolean is_double(arg);

element arg;
```

Description

The `is_double` function will return `true` if the argument is a double value; it will return `false` otherwise.

Name

is_list — checks to see if the argument is a double

Synopsis

```
boolean is_list(arg);

element arg;
```

Description

The `is_list` function will return `true` if the argument is a list; it will return `false` otherwise.

Name

is_long — checks to see if the argument is a long

Synopsis

```
boolean is_long(arg);

element arg;
```

Description

The `is_long` function will return `true` if the argument is a long value; it will return `false` otherwise.

Name

is_nlist — checks to see if the argument is an nlist

Synopsis

```
boolean is_nlist(arg);

element arg;
```

Description

The `is_nlist` function will return `true` if the argument is an nlist; it will return `false` otherwise.

Name

is_null — checks to see if the argument is `null`

Synopsis

```
boolean is_null(arg);

element arg;
```

Description

The `is_null` function will return a `true` value if the argument is `null`; it will return `false` otherwise.

Name

is_number — checks to see if the argument is a number

Synopsis

```
boolean is_number(arg);

element arg;
```

Description

The `is_number` function will return a `true` value if the argument is a number (long or double); it will return `false` otherwise.

Name

is_property — checks to see if the argument is a property

Synopsis

```
boolean is_property(arg);

element arg;
```

Description

The `is_property` function will return a `true` value if the argument is a property (atomic value); it will return `false` otherwise.

Name

is_resource — checks to see if the argument is a resource

Synopsis

```
boolean is_resource(arg);

element arg;
```

Description

The `is_resource` function will return a `true` value if the argument is a resource (collection); it will return `false` otherwise.

Name

is_string — checks to see if the argument is a string

Synopsis

```
boolean is_string(arg);

element arg;
```

Description

The `is_string` function will return `true` if the argument is a string value; it will return `false` otherwise.

Name

key — returns name of child based on the index

Synopsis

```
string key(resource, index);

nlist resource;
long index;
```

Description

This function returns the name of the child identified by its index, this can be used to iterate through all the children of an nlist. The index corresponds to the key's position in the list of all keys, sorted in lexical order. The first index is 0.

```
'/table' = nlist('red', 0xf00, 'green', 0x0f0, 'blue', 0x00f);

'/keys' = {

  tbl = value('/table');
  res = '';
  len = length(tbl);
  idx = 0;
  while (idx < len) {
    res = res + key(tbl, idx) + ' ';
    idx = idx + 1;
  };

  if (length(res) > 0) splice(res, -1, 1);
  return(res);
};
# /keys will be the string 'blue green red '
```

Name

length — returns size of a string or resource

Synopsis

```
long length(str);

string str;

long length(res);

resource res;
```

Description

Returns the size of the given string or the number of children of the given resource.

Name

list — create a new list consisting of the function arguments

Synopsis

```
list list(elem, );

element elem;
...;
```

Description

Returns a newly created list containing the function arguments.

```
# creates an empty list
'/empty' = list();

# define list of two DNS servers
'/dns' = list('137.138.16.5', '137.138.17.6');
```

Name

match — checks if a regular expression matches a string

Synopsis

```
boolean match(target, regex);

string target;
string regex;
```

Description

This function checks if the given string matches the regular expression.

```
# device_t is a string that can only be "disk", "cd" or "net"
type device_t = string with match(self, '^(disk|cd|net)$');
```

Name

matches — checks if a regular expression matches a string

Synopsis

```
string[] matches(target, regex);

string target;
string regex;
```

Description

This function matches the given string against the regular expression and returns the list of captured substrings, the first one (at index 0) being the complete matched string.

```
# IPv4 address in dotted number notation
type ipv4 = string with {
  result = matches(self, '^(\d+)\.(\d+)\.(\d+)\.(\d+)$');
  if (length(result) == 0)
  return("bad string");
  i = 1;
  while (i <= 4) {
    x = to_long(result[i]);
    if (x > 255) return("chunk " + to_string(i) + " too big: " + result[i]);
    i = i + 1;
  };
  return(true);
};
```

Name

merge — combine two resources into a single one

Synopsis

```
resource merge(res1, res2, );

resource res1;
resource res2;
...;
```

Description

This function returns the resource which combines the resources given as arguments, all of which must be of the same type: either all lists or all nlists. If more than one nlist has a child of the same name, an error occurs.

```
# /z will contain the list 'a', 'b', 'c', 'd', 'e'
'/x' = list('a', 'b', 'c');
'/y' = list('d', 'e');
'/z' = merge (value('/x'), value('/y'));
```

Name

nlist — create an nlist from the arguments

Synopsis

```
nlist nlist(key, property, );

string key;
element property;
...;
```

Description

The `nlist` function returns a new nlist consisting of the passed arguments; the arguments must be key value pairs. All of the keys must be strings and have values that are legal hash keys.

```
# resulting nlist associates name with long value
'/result' = nlist(
  'one', 1,
  'two', 2,
  'three', 3,
};
```

Name

next — increment iterator over a resource

Synopsis

```
boolean next(res, key, value);

resource res;
identifier key;
identifier value;
```

Description

This function increments the iterator associated with `res` so that it points to the next child element. The key and value of the next child are stored in the named variables `key` and `value`, either of which could be `undef`. The function returns `true` if the child exists, or `false` otherwise.

Name

path_exists — determines if a path exists

Synopsis

```
boolean path_exists(path);

string path;
```

Description

This function will return a boolean indicating whether the given path exists. The path must be an absolute or external path. This function should be used in preference to the `exists` function to avoid an ambiguity in handling the argument to `exists` as a path or variable reference.

Name

prepend — adds a value to the beginning of a list

Synopsis

```
list prepend(value);

element value;

list prepend(target, value);

list target;
element value;

list prepend(target, value);

variable_reference target;
element value;
```

Description

The `prepend` function will add the given value to the beginning of the target list. There are three variants of this function. For all of the variants, an explicit `null` value is illegal and will terminate the compilation with an error.

The first variant takes a single argument and always operates on `SELF`. It will directly modify the value of `SELF` and give the modified list (`SELF`) as the return value. If `SELF` does not exist, is `undef`, or is `null`, then an empty list will be created and the given value prepended to that list. If `SELF` exists but is not a list, an error will terminate the compilation. This variant cannot be used to create a compile-time constant.

```
# /result will have the values 2 and 1 in that order
'/result' = list(1);
'/result' = prepend(2);
```

The second variant takes two arguments. The first argument is a list value, either a literal list value or a list calculated from a DML block. This version will create a copy of the given list and prepend the given value to the copy. The modified copy is returned. If the target is not a list, then an error will terminate the compilation. This variant can be used to create a compile-time constant as long as the target expression does not reference information outside of the DML block by using, for example, the `value` function.

```
# /result will have the values 2 and 1 in that order
# /x will only have the value 1
'/x' = list(1);
'/result' = prepend(value('/x'), 2);
```

The third variant also takes two arguments, where the first value is a variable reference. This variant will take precedence over the second variant. This variant will directly modify the referenced variable and return the modified list. If the referenced variable does not exist, it will be created. As for the other forms, if the referenced target exists and is not a list, then an error will terminate the compilation. `SELF` or descendants of `SELF` can be used as the target. This variant can be used to create a compile-time constant if the referenced variable is an *existing* local variable. Referencing a global variable (except via `SELF`) is not permitted as modifying global variables from within a DML block is forbidden.

```
# /result will have the values 2 and 1 in that order
'/result' = {
  prepend(x, 1); # will create local variable x
```

```
  prepend(x, 2);
};
```

Name

replace — replace all occurrences of a regular expression

Synopsis

```
string replace(regex, repl,  target);

string regex;
string repl;
string target;
```

Description

The `replace` function will replace all occurrences of the given regular expression with the replacement string. The regular expression is specified using the standard pan regular expression syntax. The replacement string may contain references to groups identified within the regular expression. The group references are indicated with a dollar sign ($) followed by the group number. A literal dollar sign can be obtained by preceding it with a backslash.

Name

return — exit DML block with given value

Synopsis

```
element return(value);

element value;
```

Description

This function interrupts the processing of the current DML block and returns from it with the given value. This is often used in user-defined functions.

```
function facto = {
  if (ARGV[0] < 2) return(1);
  return(ARGV[0] * facto(ARGV[0] - 1));
};
```

Name

splice — insert string or list into another

Synopsis

```
string splice(str, start, length, repl);

string str;
long start;
long length;
string repl;

list splice(list, start, length, repl);

list list;
long start;
long length;
list repl;
```

Description

The first form of this function deletes the substring identified by `start` and `length` and, if a fourth argument is given, inserts `repl`.

```
'/s1' = splice('abcde', 2, 0, '12');  # ab12cde
'/s2' = splice('abcde', -2, 1);       # abce
'/s3' = splice('abcde', 2, 2, 'XXX'); # abXXXe
```

The second form of this function deletes the children of the given list identified by `start` and `length` and, if a fourth argument is given, replaces them with the contents of `repl`.

```
# will be the list 'a', 'b', 1, 2, 'c', 'd', 'e'
'/l1' = splice(list('a','b','c','d','e'), 2, 0, list(1,2));

# will be the list 'a', 'b', 'c', 'e'
'/l2' = splice(list('a','b','c','d','e'), -2, 1);

# will be the list 'a', 'b', 'XXX', 'e'
'/l3' = splice(list('a','b','c','d','e'), 2, 2, list('XXX'));
```

Important

This function will *not* modify the arguments directly. Instead a copy of the input string or list is created, modified, and returned by the function. If you ignore the return value, then the function call will have no effect.

Name

split — split a string using a regular expression

Synopsis

```
string[] split(regex, target);

string regex;
string target;

string[] split(regex, limit, target);

string regex;
long limit;
string target;
```

Description

The `split` function will split the *`target`* string around matches of the given regular expression. The regular expression is specified using the standard pan regular expression syntax. If the *`limit`* parameter is not specified, a default value of 0 is used. If the *`limit`* parameter is negative, then the function will match all occurrences of the regular expression and return the result. A value of 0 will do the same, except that empty strings at the end of the sequence will be removed. A positive value will return an array with at most *`limit`* entries. That is, the regular expression will be matched at most *`limit`*-1 times; the unmatched part of the string will be returned in the last element of the list.

Name

substr — extract a substring from a string

Synopsis

```
string substr(target, start);

string target;
long start;

string substr(target, start, length);

string target;
long start;
long length;
```

Description

This function returns the part of the given string characterised by its *`start`* position (starting from 0) and its *`length`*. If *`length`* is omitted, returns everything to the end of the string. If *`start`* is negative, starts that far from the end of the string; if *`length`* is negative, leaves that many characters off the end of the string.

```
"/s1" = substr("abcdef", 2); # cdef
"/s2" = substr("abcdef", 1, 1); # b
"/s3" = substr("abcdef", 1, -1); # bcde
"/s4" = substr("abcdef", -4); # cdef
"/s5" = substr("abcdef", -4, 1); # c
"/s6" = substr("abcdef", -4, -1); # cde
```

Name

to_boolean — convert argument to a boolean value

Synopsis

```
boolean to_boolean(prop);

property prop;
```

Description

This function converts the given property into a boolean value. The numeric values 0 and 0.0 are considered `false`; other numbers, `true`. The empty string and the string "false" (ignoring case) will return `false`; all other strings will return `true`. The function will not accept resources.

Name

to_double — convert argument to a double value

Synopsis

```
double to_double(prop);

property prop;
```

Description

This function converts the given property into a double.

If the argument is a string, then the string will be parsed to determine the double value. Any valid literal double syntax can be used. Strings that do not represent a valid double value will cause a fatal error.

If the argument is a boolean, then the function will return `0.0` or `1.0` depending on whether the boolean value is `false` or `true`, respectively.

If the argument is a long, then the corresponding double value will be returned.

If the argument is a double, then the value is returned directly.

Name

to_long — convert argument to a long value

Synopsis

```
long to_long(prop);

property prop;
```

Description

This function converts the given property into a long value.

If the argument is a string, then the string will be parsed to determine the long value. The string may represent a long value as an octal, decimal, or hexadecimal value. The syntax is exactly the same as for specifying literal long values. String values that cannot be parsed as a long value will result in an error.

If the argument is a boolean, then the return value will be either `0` or `1` depending on whether the boolean is `false` or `true`, respectively.

If the argument is a double value, then the double value is rounded to the nearest long value.

If the argument is a long value, it is returned directly.

Name

to_lowercase — change all uppercase letters to lowercase

Synopsis

```
string to_lowercase(target);

string target;
```

Description

The `to_lowercase` function will convert all uppercase letters in the *`target`* to lowercase. The United States (US) locale is forced for the conversion to guarantee consistent behavior independent of the current default locale.

Name

to_string — convert argument to a string value

Synopsis

```
string to_string(elem);

element elem;
```

Description

This function will convert the argument into a string. The function will create a reasonable human-readable representation of all data types, including lists and nlists.

Name

to_uppercase — change all lowercase letters to uppercase

Synopsis

```
string to_uppercase(target);

string target;
```

Description

The to_uppercase function will convert all lowercase letters in the target to uppercase. The United States (US) locale is forced for the conversion to guarantee consistent behavior independent of the current default locale.

Name

traceback — print message and traceback to console

Synopsis

```
string traceback(msg);

string msg;
```

Description

Prints the argument and a traceback from the current execution point to the console (stderr). Value returned is the argument. An argument that is not a string will cause a fatal error; the traceback will still be printed. This may be selectively enabled or disabled via a compiler option. See the compiler manual for details.

Name

unescape — replaces escaped characters with ASCII characters

Synopsis

```
string unescape(str);

string str;
```

Description

This function replaces escaped characters in the given string `str` to get back the original string. This is the inverse of the `escape` function.

Name

value — retrieve a value specified by a path

Synopsis

```
element value(path);

string path;
```

Description

This function returns the element identified by the given path, which can be an external path. An error occurs if there is no such element.

```
# /y will be 200
'/x' = 100;
'/y' = 2 * value('/x');
```